Contributions to the Formal Verification of Arithmetic Algorithms

Erik Martin-Dorel

N° d'ordre : 742 N° attribué par la bibliothèque : 2012ENSL0742

École Normale Supérieure de Lyon
Laboratoire de l'Informatique du Parallélisme

THÈSE

présentée par

Érik Martin-Dorel

pour l'obtention du grade de

Docteur de l'École Normale Supérieure de Lyon – Université de Lyon

spécialité : Informatique

au titre de l'École Doctorale de Mathématiques et d'Informatique Fondamentale de Lyon

Contributions to the Formal Verification of Arithmetic Algorithms

Directeur de thèse : Jean-Michel Muller
Co-encadrant de thèse : Micaela Mayero

Soutenue publiquement le 26 septembre 2012

Après avis des rapporteurs :
Yves Bertot,
John Harrison,
Frédéric Messine

Devant le jury composé de :
Paul Zimmermann, *président*,
Yves Bertot,
Sylvie Boldo,
Micaela Mayero,
Frédéric Messine,
Jean-Michel Muller

Contents

Remerciements

Tout d'abord je tiens à remercier mes directeurs de thèse Micaela Mayero et Jean-Michel Muller pour leur soutien constant tout au long de ces trois années, pour leur générosité bienveillante et leurs encouragements pour aller toujours plus loin.

Merci à Yves Bertot, John Harrison et Frédéric Messine d'avoir accepté d'être rapporteurs de cette thèse. Merci à Paul Zimmermann d'avoir accepté de présider mon jury de soutenance. Merci à Sylvie Boldo pour sa présence dans le jury. Merci à tous les membres de mon jury pour leurs remarques constructives.

Merci à Nicolas Brisebarre pour son engagement dans l'action CoqApprox et merci aux membres du projet ANR TaMaDi pour les collaborations intéressantes et les discussions que nous avons eues lors de nos multiples réunions, notamment avec Guillaume H., Guillaume M., Ioana, Jean-Michel, Laurence, Laurent, Micaela, Mioara et Nicolas.

Merci à mes encadrants de stage de Master 2 recherche Annick Truffert et Marc Daumas, qui m'ont initié aux méthodes formelles et donné le goût de la recherche.

Merci à mes coburaux Adrien, Christophe, David, Ioana, Ivan, Nicolas, Philippe, Serge et Xavier, à l'équipe-projet Arénaire/AriC et à tous les membres du laboratoire LIP de l'ENS de Lyon que j'ai côtoyés durant ces années. Merci aux ingénieurs et assistant(e)s du LIP pour leur gentillesse et leur dévouement.

Merci à Assia Mahboubi pour son code LaTeX fondé sur l'extension `listings` pour la coloration syntaxique du code Coq/SSReflect.

Merci à mes parents Annie et Gérard pour leur immense affection.

Chapter 1

Introduction

Floating-point arithmetic is by far the most frequently used way to deal with real numbers on computers: from mathematical software to safety-critical systems, floating-point operators and functions are ubiquitous. However, the various sets of floating-point numbers we can consider are not closed under these operations. For instance, we can notice that for a mere addition of two floating-point numbers of a given precision, the exact mathematical result is in general not representable with the same precision. Hence the need to perform some approximations, which leads to the notion of rounding error. However, just requiring that operations commit a rounding error bounded by a given threshold would lead to quite a weakly-specified arithmetic. In particular, this lack of specification would have a serious impact on the reproducibility of floating-point computations, as well as on the accuracy of the obtained results.

In contrast, the IEEE 754 standard for floating-point arithmetic, published in 1985, requires that the five operations $+, -, \times, \div, \sqrt{\cdot}$ should be *correctly rounded*, that is to say, the computed result must always be equal to the rounding of the exact value. This contributed to a certain level of portability and provability of floating-point algorithms. Until the 2008 revision of this standard, there was no such analogous requirement for the standard functions. The main impediment for this was the so-called *Table Maker's Dilemma* (TMD), which is the problem that occurs when one tries to choose, among a discrete set, the element that is closest to an exact mathematical result, only accessible by means of approximations. This problem actually arose a long time before the invention of computers, namely when some mathematicians designed *tables* of values for mathematical functions, such as tables of logarithms.

Roughly speaking, solving the TMD problem for an elementary function f and a given rounding mode consists of finding the floating-point numbers x such that $f(x)$ is closest to the discontinuity points of the considered rounding function. These numbers, called *hardest-to-round cases*, or simply *worst cases*, play a central role for implementing such a function f with correct rounding, since they are the inputs for which $f(x)$ is "hardest to round." The TMD can thus be viewed as a discrete, global optimization problem and can be efficiently solved in a naive way (by evaluating the function with large precision at each floating-point input) only for 32-bit arithmetic, where the number of cases to test is at most 2^{32}.

In order to address the TMD for higher precisions, two involved algorithms

by Lefèvre and by Stehlé-Lefèvre-Zimmermann (SLZ) have been designed, both starting by splitting the domain under study in small sub-intervals, then replacing the function f with an approximation polynomial P over each sub-interval. These two algorithms are based on long and complex calculations (several cumulated years×CPU), and their only output as of today is a list of hard-to-round cases and a claim that the worst one is in that list, for which trust in the algorithm, but also in the implementation, is required. This situation is thus somewhat unsatisfactory: the initial motivation for the TMD problem is to provide strong guarantees for numerical computations, but the algorithms to solve it require a large degree of trust in highly optimized implementations of complicated algorithms. To overcome this situation, we rely on so-called *formal methods*.

Formal methods gather a wide spectrum of mathematically-based techniques for specifying and verifying software or hardware systems. They are used in areas where errors can cause loss of life or significant financial damage, as well as in areas where common misunderstandings can falsify key assumptions. This includes the verification of critical systems such as those involved in aeronautics [129], where safety and security are a major concern. Formal methods have also been successfully used in floating-point arithmetic [149, 71, 72, 109, 45, 16, 22, 130, 44]: these references present works that use a formal proof assistant such as ACL2 [96], HOL Light [73], Coq [10] and PVS [139]. In this thesis, we will rely on the Coq formal proof assistant.

Basically, a proof assistant is a piece of software that allows one to encode formal specifications of programs and/or mathematical theorems in a formal language, then develop proofs that the considered programs meet their specifications. The use of such a tool is in general very expensive, since the verification process requires that the user scrutinizes many details in the proofs that would typically be left implicit in a pen-and-paper proof. Yet this has the clear benefit to force the explicit statement of all hypotheses, and thereby to avoid any incorrect use of theorems. Moreover, the fact that the proofs are mechanically checked by the computer using a proof assistant, provides strong guarantees on these proofs, as lengthy as they might be.

Contributions

In this thesis, we thus focus on the formal verification of results related to floating-point arithmetic, using the Coq formal proof assistant. In particular, we address the following problems:

- When implementing an elementary function on machines (with or without correct rounding), one of the main steps consists of determining an approximation of the function by a polynomial that has values very close to those of the considered function. It is then important to *properly handle the error* introduced by such an approximation. Indeed, reliable implementations of functions, as well as the initial step of the Lefèvre and SLZ algorithms for solving the TMD, require the specified bounds on the approximation error to be correct. To this end, we develop a formal library of *Rigorous Polynomial Approximation* in Coq. A key feature of this library CoqApprox is the ability to compute an approximation polynomial P and an *interval remainder* $\mathbf{\Delta}$ that safely bounds the approximation

error, within the proof assistant. This formalization led us to implement some symbolic-numeric techniques [88] in a formal setting, relying on the CoqInterval library [117] for *interval arithmetic* in Coq. The first part of this work resulted in a publication in the proceedings of the NASA Formal Methods 2012 conference [28].

- Then, in order to provide strong guarantees on the results produced by the SLZ algorithmic chain, we focus on the notion of *certificate*. The idea is that it is possible to produce "logs" for the execution of the SLZ algorithm, stored in the form of a certificate that can be verified *a posteriori* by a certificate checker, this latter being itself formally verified by the proof assistant. This approach led us to develop a Coq library that gathers a formalization of Hensel's lemma for both univariate and bivariate cases, along with some formally verified certificate checkers that can be executed within the logic of Coq, for verifying instances of small-integral-roots problems. In particular, the CoqHensel library so obtained provides a certificate checker for what is called the *Integer Small Value Problem* in [160], which is the problem to be dealt with when formally verifying the results of the SLZ algorithm.

- Then, solving the TMD for a bivariate function such as $(x, y) \mapsto \sqrt{x^2 + y^2}$ is often a challenge, especially since the potential number of cases to test in an enumeration strategy for finding the hardest-to-round cases is squared, compared with a univariate function. Moreover, it has been shown that this function has many *exact cases,* that is to say, a large amount of floating-point inputs (x, y) have a 2D norm that is exactly a discontinuity point of the considered rounding mode [87]. Consequently, it is not practically feasible to filter all this exceptional data beforehand when implementing a 2D norm with correct rounding. Focusing on the mathematical proof of some "*augmented-precision algorithms*" for computing square roots and 2D norms, we show how we can solve the TMD to obtain correctly-rounded 2D norms. This work has been published in the proceedings of the IEEE ARITH 2011 conference [29].

- Finally, we focus on the *double rounding* phenomenon, which typically occurs in architectures where several precisions are available and when the rounding-to-nearest mode is used by default. We thus study the potential influence of double roundings on a few usual summation algorithms in radix 2. Then relying on the Flocq library for floating-point arithmetic [22], we formalize in Coq all the corresponding proofs related to the Fast2Sum algorithm. This led us to develop some new support results in the formalism of Flocq, notably on *midpoints*, which are values that are exactly halfway between consecutive floating-point numbers. (In other words, they correspond to the discontinuity points of rounding-to-nearest modes.)

Outline of the thesis

The manuscript is organized as follows:

- In Chapter 2, we recall the required background on floating-point arithmetic and propose a mathematically-precise formulation of the TMD problem at stake;
- In Chapter 3, we give a general survey of the Coq proof assistant and describe the main concepts involved in an everyday use of Coq, most of them being required for this manuscript to be self-contained. The last section is devoted to a summary of the main Coq libraries that are related to our formal developments;
- In Chapter 4, we present our formalization of *Rigorous Polynomial Approximation* in Coq;
- In Chapter 5, we recall some key algorithmic features of *Hensel lifting,* and present our formalization of some "*integral-roots-certificates*" in Coq;
- In Chapter 6, we present the mathematical proof of some "*augmented-precision algorithms*" that can be used to get correctly-rounded 2D norms;
- In Chapter 7, we study the behavior of some summation algorithms with *double roundings,* and present our formalization of the results on Fast2Sum;
- Finally Chapter 8 concludes the manuscript.

Part I

Prerequisites

Chapter 2

Floating-Point Arithmetic

This chapter presents the required background on floating-point arithmetic and gives some mathematically-precise definitions related to the *Table Maker's Dilemma,* which we supplement with some proofs when it contributes to the clarity of exposition.

2.1 Floating-Point Representations of Numbers

In this section we start by recalling some basic definitions related to the notion of floating-point (FP) numbers, which is by far the most frequently used way to represent real numbers on machines.

Definition 2.1 (FP format). A *floating-point format* is partly defined by four integers

- $\beta \geqslant 2$, the *radix* (also known as *base*, typically 2 or 10);
- $p \geqslant 2$, the *precision* (roughly speaking, the number of "significant digits");
- $e_{\min}$ and $e_{\max}$, the *extremal exponents* (satisfying $e_{\min} < 0 < |e_{\min}| \leqslant e_{\max}$ in practice).

Note that we have written "partly defined" because in practice, some numerical data such as infinities as well as undefined results need to be represented.

Definition 2.2 (FP number). We say that a given real x is a *finite FP number* in a given FP format if there exists at least one pair (M, e) of signed integers satisfying

$$x = M \cdot \beta^{e-p+1} \tag{2.1a}$$

$$|M| < \beta^p \tag{2.1b}$$

$$e_{\min} \leqslant e \leqslant e_{\max}. \tag{2.1c}$$

Note that (2.1a) implies that the finite FP numbers form a subset of the rational numbers, since β, p, M, e are integers.

Note also that the representation (M, e) of a FP number is not necessarily unique. For example, in radix 10 and precision 5, the number 1013 can be represented either by 1013×10^0 or by 10130×10^{-1}, since both 1013 and 10130 are less than 10^5.

Definition 2.3 (Integral significand, quantum exponent). Assuming e is the smallest integer satisfying (2.1) in Definition 2.2, we will call M *the integral significand* of x, and $e-p+1$ *the quantum exponent* of x, borrowing the term chosen in [128, p. 14].

We recall another useful (and equivalent) way of defining a FP number:

Definition 2.4 (FP number, second version). We say that $x \in \mathbb{R}$ is a *finite FP number* if there exists a triple (s, m, e) satisfying

$$x = s \times m \times \beta^e \tag{2.2a}$$
$$s \in \{+1, -1\} \tag{2.2b}$$
$$m = (d_0.d_1 \cdots d_{p-1})_\beta \text{ , i.e.,}$$
$$m = \textstyle\sum_{i=0}^{p-1} d_i \beta^{-i} \quad \text{with} \quad \forall i \in [\![0, p[\![\,,\ d_i \in [\![0, \beta[\![\tag{2.2c}$$
$$e \in [\![e_{\min}, e_{\max}]\!] \,. \tag{2.2d}$$

Here again, the representation is not necessarily unique, but we may define:

Definition 2.5 (Significand and exponent). Assuming e is the smallest integer satisfying (2.2) in Definition 2.4, we will call m *the significand* of x, and e *the exponent* of x.

In the previous example of radix-10 and precision-5 format, we have $1013 = 1.013 \times 10^3$, so that the exponent of 1013 is 3.

The link between Definitions 2.3 and 2.5 follows from the following simple formula:

$$m = \frac{|M|}{\beta^{p-1}} \ (\in \mathbb{Q}), \tag{2.3}$$

so that condition (2.1b) becomes

$$0 \leqslant m < \beta. \tag{2.4}$$

When defining the (quantum) exponent and the (integral) significand of a FP number x, we have mentioned the fact that the FP representations are not necessarily unique, hence the need of a convention when a "canonical" representation is desired:

Definition 2.6 (Normalized representations). Among the various representations of a finite FP number x that are valid in the sense of Definition 2.2 (resp. Definition 2.4), the one that has the smallest possible exponent is called *the normalized representation* of x. When looking closely at a normalized representation, we can distinguish between two different kinds of FP numbers:

- the representations that satisfy $\beta^{p-1} \leqslant |M| < \beta^p$, i.e., $1 \leqslant m < \beta$ (the first digit d_0 involved in (2.2c) is nonzero): we say that x is a *normal* FP number;
- and the representations such that $|M| < \beta^{p-1}$, i.e., $0 \leqslant m < 1$: we say that x is a *subnormal* FP number and this implies that the exponent e of x satisfies $e = e_{\min}$.

Note that this distinction between normal and subnormal numbers is strongly linked to the presence of Constraint (2.2d) (resp. (2.1c)): for FP numbers with unbounded exponents, the notion of subnormal number will be quite meaningless, given that any *nonzero* FP number would have a normal representation.

Although the definition of "zero" is often presented apart the other finite FP numbers, we consider here for the sake of simplicity that it is a subnormal number that admits two representations $+0$ and -0, which correspond to triples $(s, m, e) = (+1, 0, e_{\min})$ and $(-1, 0, e_{\min})$.

A peculiarity of normal representations in radix 2 is that the first bit of the significand is always 1, so that it may be omitted in the encoding of these numbers. This is often called the implicit bit (or hidden bit) convention.

Remark 2.1 (Gradual underflow). From an architectural point of view, the handling of subnormal numbers can sometimes be a difficult feature to implement [152]. However as shown by Figure 2.1, the lack of support for subnormal numbers can falsify some key assumptions and trigger some unexpected arithmetic behavior: in particular if $a \neq b$ holds (for two finite FP numbers a and b), this would not imply that the computation of $b - a$ is nonzero, which may thus lead to a division-by-zero error in a program that relies on this kind of test. Contrastingly, the availability of subnormal numbers avoids such pitfalls: the result of a FP computation may happen to be a nonzero subnormal FP number, and this phenomenon is usually called *gradual underflow* [38, 93].

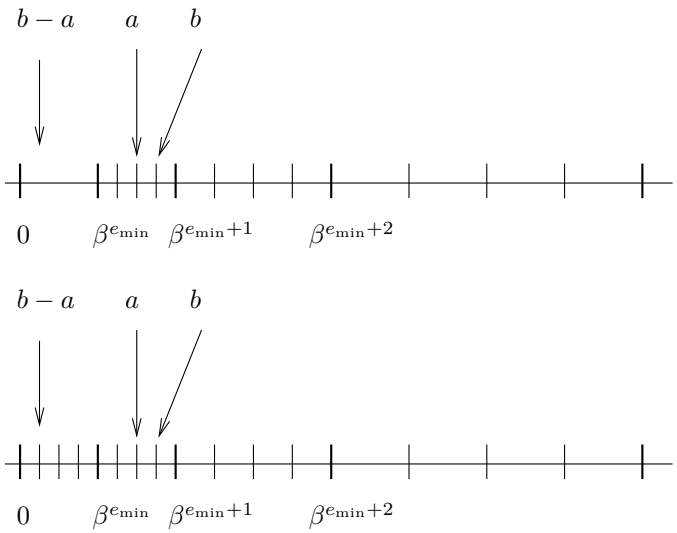

Figure 2.1 – (Taken from [128], with permission) *The* positive FP numbers *in the toy FP format* $\beta = 2$ *and* $p = 3$*. Above: normal FP numbers only. In that set,* $b - a$ *cannot be represented, so that the computation of* $b - a$ *in round-to-nearest mode (see Section 2.2) will return* 0*. Below: the subnormal numbers are included in the set of FP numbers.*

Definition 2.7 (Special FP values). Beyond the previously defined finite FP numbers, we usually define some special FP values, of which we will briefly summarize the usefulness:

- the infinite FP numbers $+\infty$ and $-\infty$ stand for numbers whose magnitude exceeds the one of the largest finite FP number as well as for "exact infinities": their availability somewhat compensates the fact that the exponent range is limited, and the arithmetic operations take these values into account. For example, we have $1/(+\infty) = +0$ and $1/(-\infty) = -0$, and conversely $1/(+0) = +\infty$ and $1/(-0) = -\infty$;

- NaN (Not-a-Number) basically encodes the fact that an invalid operation occurred (e.g., $(+\infty)-(+\infty)$, $(+0)\times(+\infty)$, $(+0)/(+0)$ and $(+\infty)/(+\infty)$ will return NaN), and this special value will be systematically propagated by further operations (e.g., $2 \times \mathrm{NaN} + 1 = \mathrm{NaN}$).

In the sequel, we will denote by $\mathbb{F}(\beta, p, e_{\min}, e_{\max})$ the set of finite FP numbers in radix β, precision p, with an exponent range of $[\![e_{\min}, e_{\max}]\!]$. Relaxing the constraint on $e_{\max}$, we also introduce the notation $\mathbb{F}(\beta, p, e_{\min})$ to denote the set of finite FP numbers with exponent $e \geqslant e_{\min}$, while we will use the notation $\mathbb{F}(\beta, p)$ to denote the set of FP numbers with no constraint at all on the exponent—we recall that in this context all nonzero FP numbers have a *normal* representation.

Definition 2.8 (Infinitely precise significand). Finally, for any nonzero real number x (whether or not it is a FP number), we will call *the infinitely precise significand* of x in radix β the number

$$\mathrm{ips}(x) := \frac{x}{\beta^{\lfloor \log_\beta |x| \rfloor}} \ (\in \mathbb{R}), \tag{2.5}$$

where $\beta^{\lfloor \log_\beta |x| \rfloor} \in \mathbb{Q}$ denotes the largest power of β less than or equal to $|x|$. This implies that we have

$$\forall x \in \mathbb{R}^*, \quad 1 \leqslant |\mathrm{ips}(x)| < \beta. \tag{2.6}$$

For example, the infinitely precise significand of $\frac{1}{3}$ in radix 10 is

$$\mathrm{ips}\left(\tfrac{1}{3}\right) = \frac{\frac{1}{3}}{10^{-1}} = 3.333\cdots.$$

Notice that if ever x is a normal FP number, its significand m_x will satisfy $m_x = |\mathrm{ips}(x)|$.

Remark 2.2 (Extremal FP numbers). For any FP format $\mathbb{F}(\beta, p, e_{\min}, e_{\max})$, the following FP numbers play a special role:

- $\Omega := (\beta^p - 1) \times \beta^{e_{\max}-p+1}$ is the largest finite FP number;

- $\beta^{e_{\min}}$ is the smallest positive normal FP number;

- $\alpha := \beta^{e_{\min}-p+1}$ is the smallest (strictly) positive subnormal FP number.

In particular, the set of real numbers whose absolute value lies in $[\beta^{e_{\min}}, \Omega]$ is usually called *the normal range*, while the set of real numbers whose absolute value lies in $[0, \beta^{e_{\min}}[$ is called *the subnormal range*.

2.2 Rounding Modes

In this section, we will fix one FP format, assuming $\mathbb{F} = \mathbb{F}(\beta, p, e_{\min}, e_{\max})$ (or possibly $\mathbb{F}(\beta, p, e_{\min})$ or $\mathbb{F}(\beta, p)$) and $\overline{\mathbb{F}} = \mathbb{F} \cup \{+\infty, -\infty\}$.

It should be noted that in any arbitrary FP format, the set of FP numbers is not closed under the usual arithmetic operations. Therefore, one has to define the way the exact (real-valued) operation on two given FP operands is mapped back to the FP numbers. Moreover, we will see later on that major concerns such as the reproducibility of FP calculations and the ability to develop mathematical proofs of FP algorithms rely on such a definition of rounding.

Definition 2.9 (Rounding modes). A *rounding mode* for the format $\overline{\mathbb{F}}$ (also called *rounding-direction attribute*) is a function $f : \mathbb{R} \longrightarrow \overline{\mathbb{F}}$ that is monotonically increasing and whose restriction to $\mathbb{F}$ is identity. We can mention five typical rounding modes:

round towards $-\infty$: this function (denoted by RD) maps $x \in \mathbb{R}$ to the largest FP number below x, that is $\mathrm{RD}(x) := \max_{\substack{f \in \overline{\mathbb{F}} \\ f \leqslant x}} f \in [-\infty, x]$;

round towards $+\infty$: this function (denoted by RU) maps $x \in \mathbb{R}$ to the smallest FP number above x, that is $\mathrm{RU}(x) := \min_{\substack{f \in \overline{\mathbb{F}} \\ f \geqslant x}} f \in [x, +\infty]$;

round towards zero: this function (denoted by RZ) maps $x \in \mathbb{R}$ to the FP number $\mathrm{RZ}(x) := \begin{cases} \mathrm{RD}(x) & \text{if } x \geqslant 0 \\ \mathrm{RU}(x) & \text{if } x \leqslant 0 \end{cases}$, which lies between 0 and x and satisfies $|\mathrm{RZ}(x)| \leqslant |x|$ (this corresponds to truncation);

round to nearest, ties to even: this function (denoted by RNE) maps $x \in \mathbb{R}$ to the closest FP number around x, with the convention for halfway cases to choose the FP number with an even integral significand;

round to nearest, ties to away from zero: this function (denoted by RNA) maps $x \in \mathbb{R}$ to the closest FP number around x, with the convention for halfway cases to choose the FP number with the largest integral significand (in absolute value).

We will frequently denote by RN an arbitrary round-to-nearest mode, i.e., with an arbitrary tie-breaking rule (not necessarily the one of RNE or RNA). The other rounding modes RD, RZ or RU are usually called *directed roundings*. The Figure 2.2 illustrates the rounded value of real numbers using these standard rounding modes.

As regards the notations, $\square : \mathbb{R} \longrightarrow \overline{\mathbb{F}}$ will refer to one of the directed roundings, while $\circ : \mathbb{R} \longrightarrow \overline{\mathbb{F}}$ will refer to an arbitrary rounding mode, i.e., a function satisfying

$$\forall (x, y) \in \mathbb{R}^2, \quad x \leqslant y \implies \circ(x) \leqslant \circ(y) \tag{2.7}$$

and

$$\forall x \in \mathbb{R}, \quad x \in \mathbb{F} \implies \circ(x) = x. \tag{2.8}$$

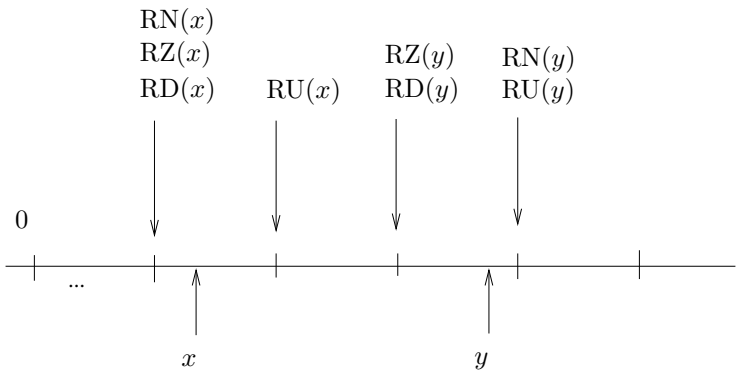

Figure 2.2 – (Taken from [128], with permission) *The standard rounding modes. Here we assume that x and y are positive numbers.*

We now define two usual notions that are very useful when dealing with error analyses:

Definition 2.10 (Unit in the last place). The *unit in last place* of any nonzero real x in a given FP format $\mathbb{F} = \mathbb{F}(\beta, p, e_{\min})$ or $\mathbb{F}(\beta, p)$ is defined by

$$\operatorname{ulp}(x) = \beta^{e_x - p + 1}, \tag{2.9}$$

where e_x is the exponent of x in $\mathbb{F}$.

Definition 2.11 (Unit roundoff). The *unit roundoff* (also known as *machine epsilon*) associated with a radix-β, precision-p FP format $\mathbb{F}$ and a rounding mode $\circ : \mathbb{R} \longrightarrow \mathbb{F}$ is defined by

$$\mathbf{u}(\circ) = \begin{cases} \operatorname{ulp}(1) \quad = \beta^{1-p} & \text{if } \circ \text{ is a directed rounding} \\ \dfrac{1}{2}\operatorname{ulp}(1) = \dfrac{1}{2}\beta^{1-p} & \text{if } \circ \text{ is a rounding-to-nearest.} \end{cases} \tag{2.10}$$

When there is no ambiguity on the considered rounding mode, we will write $\mathbf{u}$ instead of $\mathbf{u}(\circ)$.

In particular, the notion of unit roundoff appears when we want to bound the relative error due to rounding:

Property 2.1 (Relative error and standard model). *Assuming x is a real number* in the normal range[1] *of a FP format $\mathbb{F}$ with radix β and precision p, the relative error committed when rounding x satisfies:*

$$\left| \frac{x - \circ(x)}{x} \right| < \mathbf{u}, \tag{2.11}$$

as well as

$$\left| \frac{x - \circ(x)}{\circ(x)} \right| \leqslant \mathbf{u},$$

[1] implying that x and $\circ(x)$ are nonzero

or equivalently, $\exists \epsilon_1, \epsilon_2 \in \mathbb{R}$ *such that*

$$|\epsilon_1| < \mathbf{u}, \quad \circ(x) = x \cdot (1 + \epsilon_1), \tag{2.12}$$

$$|\epsilon_2| \leqslant \mathbf{u}, \quad \circ(x) = \frac{x}{1 + \epsilon_2}. \tag{2.13}$$

When replacing the $x \in \mathbb{R}$ *above with* $x \top y$ *for a given operation* $\top : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$ *and* $(x, y) \in \mathbb{F} \times \mathbb{F}$*, this property is usually known as the* "standard model" *or* "ϵ-model," *and is applicable whenever* $x \top y$ *does not underflow nor overflow.*

We can thus observe Inequality (2.11) on the example given by Figure 2.3, for which $\mathbf{u}(\mathrm{RN}) = \frac{1}{8} = 0.125$.

Proof. See for instance [79, Theorem 2.2, p. 38] for a detailed proof of (2.13) in the case of rounding-to-nearest (the strict inequality being not obvious in this case). □

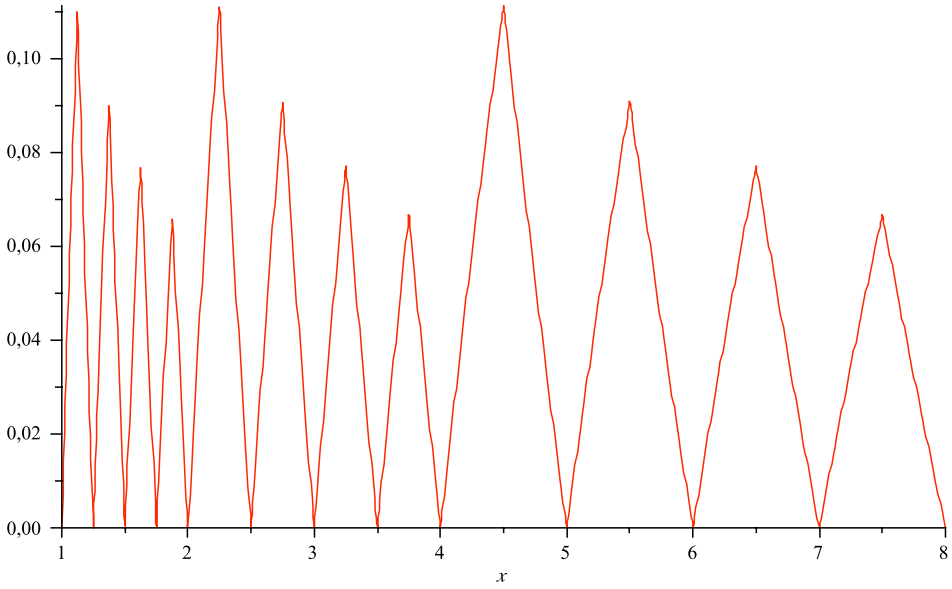

Figure 2.3 – (Taken from [128], with permission) *Relative error* $|x - \mathrm{RN}(x)| \,/\, |x|$ *that occurs when representing a real number* x *in the normal range by its nearest floating-point approximation* $\mathrm{RN}(x)$*, in the toy floating-point format* $\beta = 2$*,* $p = 3$.

Remark 2.3 (Faithful rounding). For the sake of completeness, we can mention the concept of *faithful rounding* frequently used in the FP literature to refer to an approximation of a real number x that is either $\mathrm{RD}(x)$ or $\mathrm{RU}(x)$ (the two FP numbers surrounding x). The bad side of this terminology is that a faithful *rounding* is not a monotone function, so it does not constitute a rounding mode in the sense of Definition 2.9.

Finally we introduce another definition related to rounding that is stronger than the notion of faithful rounding, and is a central notion for the present thesis:

Definition 2.12 (Correct rounding). For any real-valued function $f : D \subset \mathbb{R}^n \longrightarrow \mathbb{R}$, FP format $\mathbb{F}$ and rounding mode $\circ : \mathbb{R} \longrightarrow \mathbb{F}$, we say that $\widetilde{f} : D \cap \mathbb{F}^n \longrightarrow \mathbb{F}$ implements f *with correct rounding* if for all FP input $x \in D \cap \mathbb{F}^n$, we have $\widetilde{f}(x) = \circ(f(x))$. In other words, the computed result must always be equal to the rounding of the exact value, as if we could pre-compute $f(x)$ with an "infinite precision" before rounding.

Beyond an improvement on the **accuracy of FP computations**, correct rounding (also known as *exact rounding*) provides some significant advantages over faithful rounding, including:

reproducibility: the condition in Definition 2.12 implies that a function that is implemented with correct rounding is *fully-specified*: it is uniquely determined by the chosen FP format and rounding mode;

portability: for a given FP format and rounding mode, any IEEE-complying implementation of functions with correct rounding (either in software or in hardware) will return the same results, which greatly *improves the portability of numerical software*;

provability: furthermore, correct rounding allows one to devise some algorithms and some correctness proofs that use this specification.

2.3 The IEEE 754 Standard for FP Arithmetic

The IEEE 754 standard for FP arithmetic, first published in 1985 [84] and revised in 2008 [85], introduced precise definitions and requirements for FP implementations that paved the way for a predictable, fully-specified, FP arithmetic. In particular, it provides the following:

FP formats: it specifies the parameters $(\beta, p, e_{\min}, e_{\max})$ of standard formats that we summarize in Tables 2.1, 2.2, and 2.3, and it also defines the "physical" *encoding* for all interchange FP formats, this notion being not central for this thesis;

rounding modes: it defines the five standard rounding modes we have mentioned in Definition 2.9 (the last one RNA being new in the 2008-revision of the standard, and especially useful for decimal arithmetic);

conversions: it defines conversions between integers and FP formats or between different FP formats, as well as between FP formats and external representations as (decimal or hexadecimal) character sequences;

operations: it requires the availability of addition, subtraction, multiplication, division, square root, comparison predicates, and (since the 2008-revision of the standard) fused-multiply-add (FMA), with the requirement that all these operations shall be correctly rounded, as stated in Definition 2.12: the FMA is thus defined as the operator $(x, y, z) \in \mathbb{F}^3 \mapsto \circ(x \times y + z) \in \mathbb{F}$;

exceptions: it also defines five kinds of exceptions that shall be signaled in the following situations: *invalid operation, division by zero, overflow*, *underflow*, and *inexact* result.

Basic format	Legacy name [84]	β	p	$e_{\min}$	$e_{\max}$
binary32	single precision	2	24	-126	127
binary64	double precision	2	53	-1022	1023
binary128		2	113	-16382	16383
decimal64		10	16	-383	384
decimal128		10	34	-6143	6144

Table 2.1 – *Specification of* basic FP formats [85].

Interchange format	storage size k (bits)	radix β	precision p (digits)	$e_{\max}$
binary16	16	2	11	15
binary32	32	2	24	127
binary64	64	2	53	1023
binary128	128	2	113	16383
binary-k	$k \in 128 + 32 \cdot \mathbb{N}$	2	$k - \lfloor 4 \times \log_2(k) \rceil + 13$	$2^{k-p-1} - 1$
decimal32	32	10	7	96
decimal64	64	10	16	384
decimal128	128	10	34	6144
decimal-k	$k \in 32 \cdot \mathbb{N}^*$	10	$9 \times k/32 - 2$	$3 \times 2^{k/16+3}$

Table 2.2 – *Specification of* interchange FP formats *(we have $e_{min} = -e_{max}+1$). Notice that all basic formats are interchange ones.*

Extended format based on	β	$p \geqslant$	$e_{\max} \geqslant$
binary32	2	32	1023
binary64	2	64	16383
binary128	2	128	65535
decimal64	10	22	6144
decimal128	10	40	24576

Table 2.3 – *Specification of* extended FP formats *(we have $e_{min} = -e_{max} + 1$). A typical example of extended format that meets these requirements is given by Intel's 80-bit binary-double-extended-precision format that indeed extends the basic FP format binary64 (it has 1 sign bit, 64 bits for the significant—including the implicit bit—, and 15 bits for the exponent, so that $e_{max} = 2^{14} - 1 \geqslant 16383$). Finally note that unlike interchange formats, the IEEE 754-2008 standard does* not require any specific encoding *for extended formats.*

As far as correct rounding is concerned, the 1985 version of the standard only required that the five basic operations ($+$, $-$, $\times$, $\div$, and $\sqrt{\cdot}$) shall produce *correctly-rounded results*. This contributed to a certain level of portability and provability of FP algorithms. Until 2008, there was no such analogous requirement for standard functions. The main impediment for this was the so-called *Table Maker's Dilemma*, which constitutes the topic of the upcoming Section 2.4. The interested reader will be able to find in [85, Table 9.1, pp. 42–43] a list of the elementary functions for which the 2008 version of the standard recommends correct rounding (without requiring it however).

2.4 The Table Maker's Dilemma

Roughly speaking, implementations of a correctly-rounded elementary function often use the so-called Ziv's strategy:

1. compute a small enclosing interval for $f(x)$, of width 2^{-q};
2. if all the points of this interval round to a single FP number, this number is $f(x)$ correctly rounded;
3. otherwise, increase q, and go to step 1.

Figure 2.4 illustrates the situation where we can conclude for correctly rounding $f(x)$ at step 2, while Figure 2.5 illustrates the opposite situation.

The test corresponding to step 2 can typically be performed by checking if the *endpoints* of the small enclosing interval have the same rounded value, because roundings are monotone functions. There is also a fast testing strategy suggested by Ziv, and analyzed in detail in [46].

Yet it can be noted that this strategy will not terminate if ever $f(x)$ is a point of discontinuity of the rounding function, since in this case no open interval containing $f(x)$ will round to a single FP number. Nevertheless, some results from transcendental number theory such as Lindemann's theorem [53] can be used to establish that for a number of transcendental functions, this situation cannot occur, except for some *exceptional argument* (e.g., $\cos(0) = 1$) [165, 128].

Regardless of these possible exceptional cases, in order to make Ziv's strategy optimal, we want to go at most once through step 3, and in this situation set q to a precision that is large enough to guarantee that the test at step 2 is true, and yet not too large, since the cost of step 1 increases drastically with q. Finding the optimal value (say m) of this "large enough" precision is the so-called *Table Maker's Dilemma* (TMD), this term having been coined by Kahan [92, 90]. Finding an upper bound for m will be called the "*Approximate TMD*". Solving the TMD, for a given function f and for each considered FP format and rounding mode, means finding the *hardest-to-round* points, that is, in rounding-to-nearest, the FP numbers x such that $f(x)$ is closest to the midpoint of two consecutive FP numbers (in terms of relative error, or in terms of error in ulps).

More formally, we will give some general definitions of these key notions related to the TMD in the upcoming Section 2.4.1, while in Section 2.4.2 we will present a general scheme that may be considered once the (Approximate)-TMD has been solved.

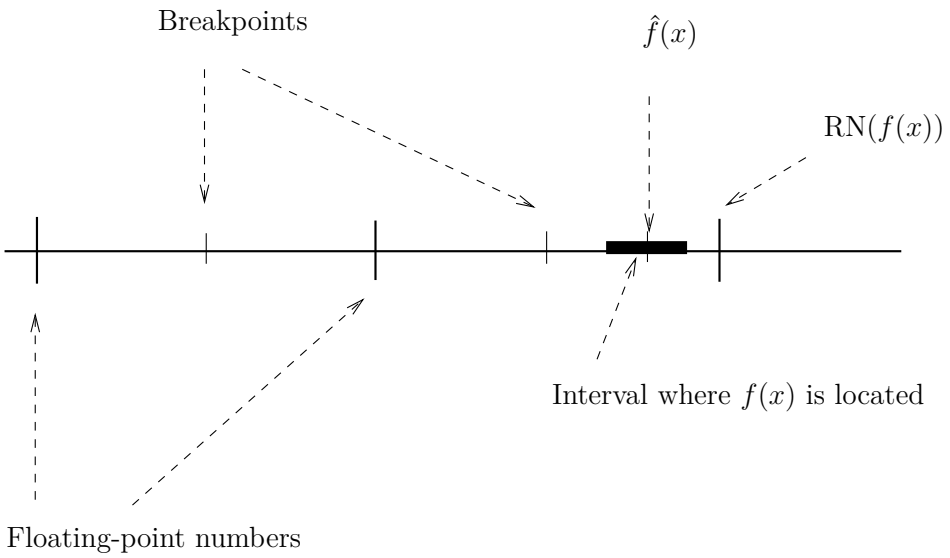

Figure 2.4 – (Taken from [128], with permission) *In this example (assuming rounding to nearest), the interval around $\hat{f}(x)$ where $f(x)$ is known to be located contains no breakpoint (see Definition 2.13). Hence,* $\mathrm{RN}(f(x)) = \mathrm{RN}(\hat{f}(x))$*, so we can safely return a correctly rounded result.*

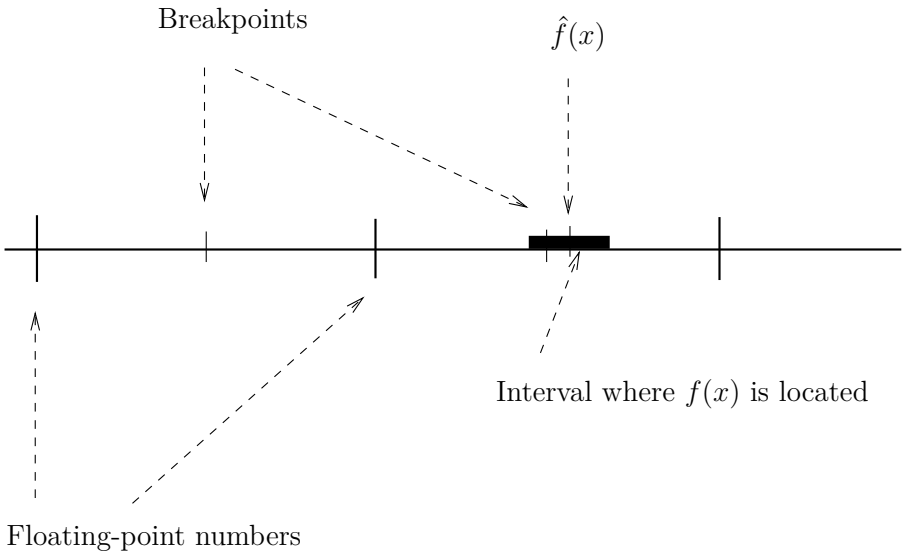

Figure 2.5 – (Taken from [128], with permission) *In this example (assuming rounding to nearest), the interval around $\hat{f}(x)$ where $f(x)$ is known to be located contains a breakpoint. We do not have enough information to provide a correctly-rounded result.*

2.4.1 Definitions and Notations

Definition 2.13 (Breakpoints and midpoints). For a given FP format $\mathbb{F}$ and rounding mode $\circ : \mathbb{R} \longrightarrow \mathbb{F}$, the discontinuities of the function $\circ$ are called *breakpoints*. We will use $\mathcal{B}(\circ)$ to denote the set of breakpoints. For the particular case of roundings-to-nearest, the breakpoints are also called *midpoints*, given that they are the real numbers exactly halfway between consecutive FP numbers.

Property 2.2 (Midpoints in even radix). *For any even radix β, the set of precision-p midpoints is a subset of the set of precision-$(p+1)$ breakpoints for directed roundings.*

The result given by Property 2.2 can be used to simplify the resolution of the TMD, by taking the directed roundings only into account without loss of generality.

Definition 2.14 (Centered modulo). We will call *centered modulo* any function $\text{cmod} : \mathbb{R} \times \mathbb{R}_+^* \longrightarrow \mathbb{R}$ satisfying

$$\forall x \in \mathbb{R}, \quad \forall v > 0, \quad \exists n \in \mathbb{Z}, \quad x \bmod v = x + nv \quad \wedge \quad |x \bmod v| \leqslant \frac{v}{2}. \tag{2.14}$$

To ensure the uniqueness of this notion, we can for example focus on the centered modulo that has values in $\left[-\frac{v}{2}, \frac{v}{2}\right[$, which amounts to considering the following definition:

$$x \bmod v := x - v \left\lfloor \frac{x}{v} + \frac{1}{2} \right\rfloor.$$

Another possibility would be to consider

$$x \bmod v := x - v \left\lceil \frac{x}{v} - \frac{1}{2} \right\rceil.$$

Note by the way that any centered modulo function satisfies

$$\forall x \in \mathbb{R}, \quad \forall v > 0, \quad |(-x) \bmod v| = |x \bmod v|.$$

Relying on this notion of centered modulo, we will precisely define the TMD problem at stake, using a formalism similar to [159, Eq. (1)] and focusing on a definition that is compatible with [128, Def. 10, p. 409].

Definition 2.15 (Distance to breakpoints). For a standard rounding mode $\circ \in \{\text{RD}, \text{RU}, \text{RZ}, \text{RN}\} : \mathbb{R} \longrightarrow \mathbb{F}(\beta, p)$, we define the normalized distance of any[2] $y \in \mathbb{R}$ to the set $\mathcal{B}(\circ)$ of breakpoints as

$$\text{dist}\,(y, \circ) = \Big|(\text{ips}(y) - \mathbf{u}(\circ)) \bmod \text{ulp}(1)\Big|, \tag{2.15}$$

which can be rewritten to

$$\text{dist}\,(y, \square) = \Big|\text{ips}(y) \bmod \text{ulp}(1)\Big| \tag{2.16}$$

in the case of directed roundings, since $\mathbf{u}(\square) = \text{ulp}(1)$.

[2] with the convention that for $y = 0$, $\text{ips}(y) = 0$

Definition 2.16 (Exact cases). We define the set of *exact cases* of $f : D \subset \mathbb{R}^n \longrightarrow \mathbb{R}$ with respect to $\mathbb{F} = \mathbb{F}(\beta, p, e_{\min}, e_{\max})$ and $\circ : \mathbb{R} \longrightarrow \mathbb{F}(\beta, p)$ as the inverse image of $\mathcal{B}(\circ)$ by f:

$$\mathrm{EC}(f, \mathbb{F}, \circ) = \mathbb{F}^n \cap f^{-1}\Big(\mathcal{B}(\circ)\Big), \tag{2.17}$$

which amounts to considering the set of FP inputs whose image by f cancels the previously defined distance:

$$\mathrm{EC}(f, \mathbb{F}, \circ) = \Big\{ x \in D \cap \mathbb{F}^n \;\Big|\; \mathrm{dist}\left(f(x), \circ\right) = 0 \Big\}. \tag{2.18}$$

Definition 2.17 (Bad cases). We define the set of (p, q)-*bad cases* of $f : D \subset \mathbb{R}^n \longrightarrow \mathbb{R}$ with respect to $\mathbb{F} = \mathbb{F}(\beta, p, e_{\min}, e_{\max})$ and $\circ : \mathbb{R} \longrightarrow \mathbb{F}(\beta, p)$ as the FP inputs whose image by f is close to a breakpoint:

$$\mathrm{BC}(f, \mathbb{F}, \circ, q) = \Big\{ x \in D \cap \mathbb{F}^n \;\Big|\; \mathrm{dist}\left(f(x), \circ\right) \leqslant \frac{1}{\beta^q} \Big\}. \tag{2.19}$$

We will occasionally refer to (p, q)-bad cases as *hard-to-round* cases, when omitting the precision p and q.

It can be noted that when q increases, the sets $\mathrm{BC}(f, \mathbb{F}, \circ, q)$ are decreasing with respect to inclusion:

$$\mathrm{BC}(f, \mathbb{F}, \circ, q) \supset \mathrm{BC}(f, \mathbb{F}, \circ, q+1) \supset \ldots, \tag{2.20}$$

yet all these sets might be nonempty, if ever one value $f(x)$ is an exact case (we recall that in this situation, Ziv's strategy would not terminate). So for solving the TMD for a given function, we have to manually exclude all these exact cases, either beforehand or after having enumerated all the (p, q)-bad cases. This explains the shape of the following definitions:

Definition 2.18 (Hardness to round). The *hardness-to-round* of $f : D \subset \mathbb{F}^n \longrightarrow \mathbb{R}$ with respect to $\mathbb{F} = \mathbb{F}(\beta, p, e_{\min}, e_{\max})$ and $\circ : \mathbb{R} \longrightarrow \mathbb{F}(\beta, p)$ can be defined as one of the following integers:

$$\mathrm{HNR}(f, \mathbb{F}, \circ) = \max\Big\{ q \in \mathbb{N} \;\Big|\; \mathrm{BC}(f, \mathbb{F}, \circ, q) \setminus \mathrm{EC}(f, \mathbb{F}, \circ) \neq \varnothing \Big\} \tag{2.21}$$

$$\mathrm{HNR}'(f, \mathbb{F}, \circ) = \min\Big\{ m \in \mathbb{N} \;\Big|\; \mathrm{BC}(f, \mathbb{F}, \circ, m) \setminus \mathrm{EC}(f, \mathbb{F}, \circ) = \varnothing \Big\}. \tag{2.22}$$

Before showing that these quantities are meaningful, we introduce another useful definition:

Definition 2.19 (Non-zero distance). The *minimum non-zero distance to breakpoints* of a given function $f : D \subset \mathbb{R}^n \longrightarrow \mathbb{R}$ with respect to a FP format $\mathbb{F} = \mathbb{F}(\beta, p, e_{\min}, e_{\max})$ and a rounding mode $\circ : \mathbb{R} \longrightarrow \mathbb{F}(\beta, p)$ is defined as:

$$\mathrm{NZD}(f, \mathbb{F}, \circ) := \min_{\substack{x \in \mathbb{F} \\ x \notin \mathrm{EC}(f, \mathbb{F}, \circ)}} \mathrm{dist}\left(f(x), \circ\right). \tag{2.23}$$

Theorem 2.1 (Hardness to round). *For any function $f : D \subset \mathbb{R}^n \longrightarrow \mathbb{R}$, FP format $\mathbb{F} = \mathbb{F}(\beta, p, e_{min}, e_{max})$ and rounding mode $\circ : \mathbb{R} \longrightarrow \mathbb{F}(\beta, p)$, if $\mathbb{F} \setminus \mathrm{EC}(f, \mathbb{F}, \circ)$ contains at least one point (which holds in all practical cases), we have the following results:*

$$\mathrm{NZD}(f, \mathbb{F}, \circ) \text{ exists}, \tag{2.24}$$

$$0 < \mathrm{NZD}(f, \mathbb{F}, \circ) \leqslant \frac{1}{2}\,\mathrm{ulp}(1), \tag{2.25}$$

$$\mathrm{HNR}(f, \mathbb{F}, \circ) \text{ and } \mathrm{HNR}'(f, \mathbb{F}, \circ) \text{ exist}, \tag{2.26}$$

$$\mathrm{HNR}(f, \mathbb{F}, \circ) = \left\lfloor -\log_\beta \mathrm{NZD}(f, \mathbb{F}, \circ) \right\rfloor, \tag{2.27}$$

$$\mathrm{HNR}(f, \mathbb{F}, \circ) \geqslant p - 1, \tag{2.28}$$

$$\mathrm{HNR}'(f, \mathbb{F}, \circ) = 1 + \mathrm{HNR}(f, \mathbb{F}, \circ), \tag{2.29}$$

$$\forall m \geqslant \mathrm{HNR}'(f, \mathbb{F}, \circ),\ \forall x \in \mathbb{F},\ \left[f(x) \in \mathcal{B}(\circ)\ \vee\ \mathrm{dist}\,(f(x), \circ) > \beta^{-m} \right] \tag{2.30}$$

Proof of Theorem 2.1.

- For (2.24): $\mathbb{F} \setminus \mathrm{EC}(f, \mathbb{F}, \circ)$ is a finite nonempty set, so it is legal to consider the "min" that appears in the definition of $\mathrm{NZD}(f, \mathbb{F}, \circ)$ in (2.23).
- For (2.25): due to the definition of EC given in (2.18), the function $x \mapsto \mathrm{dist}\,(f(x), \circ)$ is strictly positive over $\mathbb{F} \setminus \mathrm{EC}(f, \mathbb{F}, \circ)$, and so is its minimum value $\mathrm{NZD}(f, \mathbb{F}, \circ) > 0$. Moreover, using the properties of the centered modulo, we have $\forall y \in \mathbb{R},\ \mathrm{dist}\,(y, \circ) \leqslant \frac{1}{2}\,\mathrm{ulp}(1)$, implying that $\mathrm{NZD}(f, \mathbb{F}, \circ) \leqslant \frac{1}{2}\,\mathrm{ulp}(1)$.
- For (2.26): Let us define the abbreviation

  $$G_q := \mathrm{BC}(f, \mathbb{F}, \circ, q) \setminus \mathrm{EC}(f, \mathbb{F}, \circ).$$

  Let us show that $\{q \in \mathbb{N} \,|\, G_q \neq \varnothing\}$ is a nonempty upper-bounded set. First, we have assumed that $\mathbb{F} \setminus \mathrm{EC}(f, \mathbb{F}, \circ)$ contains at least one point, say $x_0 \in \mathbb{F}$. Since $x_0 \notin \mathrm{EC}(f, \mathbb{F}, \circ)$, we have $0 < \mathrm{dist}\,(f(x_0), \circ) \leqslant \frac{1}{2}\,\mathrm{ulp}(1) < 1 = \frac{1}{\beta^0}$, implying that $G_0 \neq \varnothing$. Second, we have:

  $$\begin{aligned} \forall q \in \mathbb{N},\quad G_q \neq \varnothing &\iff \exists x \in G_q \\ &\iff \exists x \in \mathbb{F},\ 0 < \mathrm{dist}\,(f(x), \circ) \leqslant \beta^{-q} \\ &\iff 0 < \mathrm{NZD}(f, \mathbb{F}, \circ) \leqslant \beta^{-q} \quad \text{by definition of "min"} \\ &\iff q \leqslant -\log_\beta \mathrm{NZD}(f, \mathbb{F}, \circ), \end{aligned} \tag{2.31}$$

  hence the existence of $\mathrm{HNR}(f, \mathbb{F}, \circ)$. Moreover the definition of $\mathrm{HNR}(f, \mathbb{F}, \circ)$ implies

  $$G_{1+\mathrm{HNR}(f, \mathbb{F}, \circ)} = \varnothing, \tag{2.32}$$

  that is to say the set $\{m \in \mathbb{N} \,|\, G_m = \varnothing\}$ is nonempty, hence the existence of $\mathrm{HNR}'(f, \mathbb{F}, \circ) = \min\{m \in \mathbb{N} \,|\, G_m = \varnothing\}$.
- For (2.27): Here we can reuse the reasoning by equivalences that led to Inequality (2.31):

  $$\begin{aligned} \mathrm{HNR}(f, \mathbb{F}, \circ) &= \max\{q \in \mathbb{N} \,|\, G_q \neq \varnothing\} \\ &= \max\left\{q \in \mathbb{N} \,\middle|\, q \leqslant -\log_\beta \mathrm{NZD}(f, \mathbb{F}, \circ)\right\} \\ &= \left\lfloor -\log_\beta \mathrm{NZD}(f, \mathbb{F}, \circ) \right\rfloor. \end{aligned}$$

- For (2.28): According to (2.25), we have

$$0 < \mathrm{NZD}(f,\mathbb{F},\circ) \leqslant \frac{1}{2}\beta^{1-p} < \beta^{1-p} \iff -\log_\beta \mathrm{NZD}(f,\mathbb{F},\circ) > p-1 \in \mathbb{Z},$$

  therefore by (2.27), we have

$$\mathrm{HNR}(f,\mathbb{F},\circ) \geqslant p-1.$$

- For (2.29): Result (2.32) directly implies that $1+\mathrm{HNR}(f,\mathbb{F},\circ) \geqslant \mathrm{HNR}'(f,\mathbb{F},\circ)$. Thus it remains to verify that

$$\mathrm{HNR}(f,\mathbb{F},\circ)+1 \leqslant \min\left\{m \in \mathbb{N} \,|\, G_m = \varnothing\right\}.$$

  If it were not true, there would exists one $m_0 \in \mathbb{N}$ such that

$$G_{m_0} = \varnothing \ \wedge \ \mathrm{HNR}(f,\mathbb{F},\circ)+1 > m_0,$$

  that is $m_0 \leqslant \mathrm{HNR}(f,\mathbb{F},\circ)$, so that Result (2.20) implies

$$G_{m_0} \supset G_{\mathrm{HNR}(f,\mathbb{F},\circ)},$$

  which contradicts the fact that $G_{\mathrm{HNR}(f,\mathbb{F},\circ)} \neq \varnothing$.

- For (2.30): For all $m \geqslant \mathrm{HNR}'(f,\mathbb{F},\circ)$, we have $G_m = \varnothing$, that is to say

$$\begin{aligned} \forall x \in \mathbb{F}, \quad x \notin G_m &\iff x \notin \mathrm{BC}(f,\mathbb{F},\circ,m) \setminus \mathrm{EC}(f,\mathbb{F},\circ) \\ &\iff x \in \mathrm{EC}(f,\mathbb{F},\circ) \ \vee \ x \notin \mathrm{BC}(f,\mathbb{F},\circ,m) \\ &\iff f(x) \in \mathcal{B}(\circ) \ \vee \ \mathrm{dist}\left(f(x),\circ\right) > \beta^{-m}. \end{aligned}$$ □

Definition 2.20 (Hardest-to-round cases). Finally, the set of *hardest-to-round cases* of $f : D \subset \mathbb{R}^n \longrightarrow \mathbb{R}$ with respect to $\mathbb{F} = \mathbb{F}(\beta,p,e_{\min},e_{\max})$ and $\circ : \mathbb{R} \longrightarrow \mathbb{F}(\beta,p)$ can be defined as the set

$$\mathrm{HRC}(f,\mathbb{F},\circ) = \mathrm{BC}\Big(f,\mathbb{F},\circ,\mathrm{HNR}(f,\mathbb{F},\circ)\Big) \setminus \mathrm{EC}(f,\mathbb{F},\circ). \tag{2.33}$$

We will sometimes use the term *worst case* as a synonym for *hardest-to-round case*.

Finally, by "solving the TMD problem for a given function, FP format and rounding mode" we will mean one of the following, interrelated assertions:

- Knowing the hardest-to-round cases $\mathrm{HRC}(f,\mathbb{F},\circ)$.
- Knowing the hardness-to-round $\mathrm{HNR}(f,\mathbb{F},\circ)$ (or $\mathrm{HNR}'(f,\mathbb{F},\circ)$);
- Knowing the nonzero distance to breakpoints $\mathrm{NZD}(f,\mathbb{F},\circ)$;

while the knowledge of a lower bound on $\mathrm{NZD}(f,\mathbb{F},\circ)$ will sometimes be referred to as the "*Approximate-TMD*" problem.

2.4.2 Practical Consequences of Solving the TMD

If we know the minimum nonzero distance between $f(x)$ and a breakpoint, or a lower bound η on this distance, then correctly rounding $f(x)$ can roughly be done as described in Algorithm 2.1. It is similar to what is called "Method B" in [165], but more general since we deal with the various rounding modes in a unified manner.

Algorithm 2.1: General scheme to provide correct rounding

Input : $f : D \subset \mathbb{R}^n \to \mathbb{R}$, $\mathbb{F} = \mathbb{F}(\beta, p, e_{\min}, e_{\max})$, $\circ : \mathbb{R} \to \mathbb{F}(\beta, p)$, $0 < \eta \leqslant \mathrm{NZD}(f, \mathbb{F}, \circ)$, $x \in D \cap \mathbb{F}^n$
Output: $\circ(f(x))$
`// Fast phase`
1 $\hat{y}_1 \leftarrow$ an approximation of $f(x)$, with an "accuracy" $\alpha_1 > 0$ slightly tighter than β^{-p}, that is $\hat{y}_1 = f(x)(1+\epsilon_1)$ with $|\epsilon_1| \leqslant \alpha_1 < \beta^{-p}$
2 **if** $\circ\left(\frac{\hat{y}_1}{1+\alpha_1}\right) = \circ\left(\frac{\hat{y}_1}{1-\alpha_1}\right)$ **then return** $\circ(\hat{y}_1)$
`// Precise phase`
3 $\alpha_2 \leftarrow \frac{4\eta}{9\beta}$
4 $\hat{y}_2 \leftarrow$ an approximation of $f(x)$, with an accuracy of α_2, that is to say $\hat{y}_2 = f(x)(1+\epsilon_2)$ with $|\epsilon_2| \leqslant \alpha_2$
5 **if** $\mathrm{dist}(\hat{y}_2, \circ) \leqslant \frac{|\mathrm{ips}(\hat{y}_2)|\,\alpha_2}{1-\alpha_2}$ **then**
6 $\quad \hat{b} \leftarrow$ the precision-p breakpoint closest to $\hat{y}_2$
7 $\quad$ **return** $\circ(\hat{b})$
8 **else**
9 $\quad$ **return** $\circ(\hat{y}_2)$
10 **end**

Proof of Algorithm 2.1. First, we notice that in the fast phase, both $\hat{y}_1$ and $f(x)$ belong to $\hat{y}_1 \cdot \left[\frac{1}{1+\alpha_1}, \frac{1}{1-\alpha_2}\right]$, so that if the endpoints of this interval round to the same value, we indeed have $\circ(f(x)) = \circ(\hat{y}_1)$, so that Line 2 is correct.
Now suppose that we have

$$\mathrm{dist}(\hat{y}_2, \circ) \leqslant \frac{|\mathrm{ips}(\hat{y}_2)|\,\alpha_2}{1-\alpha_2}.$$

and $\hat{y}_2 \neq 0$, implying $f(x) \neq 0$. Let $\hat{b}$ be the precision-p breakpoint closest to $\hat{y}_2$

and let us notate $y := f(x)$. We have:

$$\begin{aligned}
\operatorname{dist}(y,\circ) &= \tfrac{\operatorname{ulp}(1)}{\operatorname{ulp}(y)} \min_{b\in\mathcal{B}(\circ)} |y-b| \\
&\leqslant \tfrac{\operatorname{ulp}(1)}{\operatorname{ulp}(y)} \left|y-\hat{b}\right| \\
&\leqslant \tfrac{\operatorname{ulp}(1)}{\operatorname{ulp}(y)} \left(|y-\hat{y}_2| + \left|\hat{y}_2-\hat{b}\right|\right) \\
&= \tfrac{\operatorname{ulp}(1)}{\operatorname{ulp}(y)} |y|\,|\epsilon_2| + \tfrac{\operatorname{ulp}(\hat{y}_2)}{\operatorname{ulp}(y)} \tfrac{\operatorname{ulp}(1)}{\operatorname{ulp}(\hat{y}_2)} \left|\hat{y}_2-\hat{b}\right| \\
&= |\operatorname{ips}(y)|\,|\epsilon_2| + \tfrac{\operatorname{ulp}(\hat{y}_2)}{\operatorname{ulp}(y)} \operatorname{dist}(\hat{y}_2,\circ) \\
&= |\operatorname{ips}(y)|\,|\epsilon_2| + \tfrac{|\operatorname{ips}(y)|}{|y|\operatorname{ulp}(1)} \tfrac{|\hat{y}_2|\operatorname{ulp}(1)}{|\operatorname{ips}(\hat{y}_2)|} \operatorname{dist}(\hat{y}_2,\circ) \\
&= |\operatorname{ips}(y)|\,|\epsilon_2| + |\operatorname{ips}(y)|\,(1+\epsilon_2)\,\tfrac{\operatorname{dist}(\hat{y}_2,\circ)}{|\operatorname{ips}(\hat{y}_2)|} \\
&< \beta\alpha_2 + \beta\,(1+\alpha_2)\,\tfrac{\alpha_2}{1-\alpha_2} \\
&\leqslant \beta\alpha_2\left(1+\tfrac{1+\alpha_2}{1-\alpha_2}\right) \\
&\leqslant \beta\alpha_2\left(1+\tfrac{1+\frac{1}{9}}{1-\frac{1}{9}}\right) \quad \text{as } \alpha_2 = \tfrac{4}{9}\eta\tfrac{1}{\beta} \leqslant \tfrac{4}{9}\tfrac{\operatorname{ulp}(1)}{2}\tfrac{1}{2} \leqslant \tfrac{1}{9}. \\
&= \eta \leqslant \operatorname{NZD}(f,\mathbb{F},\circ),
\end{aligned} \tag{2.34}$$

hence $\operatorname{dist}(y,\circ) = 0$, that is to say, $f(x) \in \mathcal{B}(\circ)$. As a matter of fact, we have $f(x) = \hat{b}$. Indeed if it were not the case, there would exist a breakpoint $b' \in \mathcal{B}(\circ)$ such that $y = f(x) = b' \neq \hat{b}$, implying by (2.34):

$$0 < \tfrac{\operatorname{ulp}(1)}{\operatorname{ulp}(b')}\left|b'-\hat{b}\right| < \eta \leqslant \frac{\operatorname{ulp}(1)}{2},$$

which turns out to be impossible (two different breakpoints cannot be so close to each other). Consequently, we have $\circ(f(x)) = \circ(\hat{b})$, i.e., Line 7 is correct. Finally, suppose that we have

$$\operatorname{dist}(\hat{y}_2,\circ) > \frac{|\operatorname{ips}(\hat{y}_2)|\,\alpha_2}{1-\alpha_2}.$$

Let $y' \in \mathbb{R}$ be an arbitrary number such that

$$\exists \epsilon' \in \mathbb{R}, \quad y' = \frac{\hat{y}_2}{1+\epsilon'} \quad \wedge \quad |\epsilon'| \leqslant |\epsilon_2|\,.$$

We recall that we have

$$\hat{y}_2 = \tfrac{\operatorname{ips}(\hat{y}_2)\operatorname{ulp}(\hat{y}_2)}{\operatorname{ulp}(1)}.$$

Consequently,

$$\begin{aligned}
\tfrac{\operatorname{ulp}(1)}{\operatorname{ulp}(\hat{y}_2)}|y'-\hat{y}_2| &= \tfrac{\operatorname{ulp}(1)}{\operatorname{ulp}(\hat{y}_2)}|y'|\,|\epsilon'| \\
&= \tfrac{\operatorname{ulp}(1)}{\operatorname{ulp}(\hat{y}_2)}\tfrac{|\hat{y}_2|}{1+\epsilon'}|\epsilon'| \\
&= |\operatorname{ips}(\hat{y}_2)|\,\tfrac{|\epsilon'|}{1+\epsilon'} \\
&\leqslant |\operatorname{ips}(\hat{y}_2)|\,\tfrac{\alpha_2}{1-\alpha_2} \\
&< \operatorname{dist}(\hat{y}_2,\circ)\,,
\end{aligned}$$

so that there is no breakpoint in the interval $\hat{y}_2 \cdot \left[\frac{1}{1+|\epsilon_2|}, \frac{1}{1-|\epsilon_2|}\right]$, which contains both $\hat{y}_2$ and $f(x)$. Therefore, one can safely return $\circ(\hat{y}_2) = \circ(f(x))$ on Line 9.□

To sum up, a key prerequisite to ensure correct rounding at an optimal cost for the usual functions, FP formats and rounding modes consists of the knowledge of their minimum nonzero distance to breakpoints, or at least a (relatively tight) lower bound of this distance.

2.5 Algorithms to Solve the TMD

2.5.1 A Naive Algorithm

A naive method of finding the hardest-to-round points consists of evaluating the function with large precision at each FP number: this is efficient only in binary32 and decimal32 formats, where the number of cases to test is at most 2^{32}.

Going beyond these 32-bit formats requires using different algorithms. Two involved algorithms have been designed to find these hardest-to-round points. Both proceed by splitting the domain under study into small intervals, and by replacing the function over each of those intervals with a polynomial approximation, for which the TMD is solved.

2.5.2 The L Algorithm

The Lefèvre algorithm uses degree-1 polynomial approximations, and reduces the TMD problem for each of these approximations to finding the integers k that satisfy the following equation, for a given $N \in \mathbb{N}^*$, $d_0 \in \mathbb{R}^*$, and $a, b \in \mathbb{R}$ being determined by the considered degree-1 approximation:

$$\begin{cases} k \in [\![0, N[\![\\ (b - k \times a) \bmod 1 < d_0, \end{cases} \tag{2.35}$$

where $y \bmod 1 := y - \lfloor y \rfloor$ denotes the positive fractional part of y. Then the L-algorithm relies on a method that can be viewed as a variant of the Euclidean GCD algorithm (cf. Algorithm 2.2 for the subtractive version, while another version involving divisions can be found in [101]).

This algorithm made it possible to obtain hardest-to-round cases in binary64 arithmetic [102, 127].

For instance, the hardest-to-round case for the natural logarithm in the full range of binary64 is

$$t = 1.0110001010101000100001100001001101100010100110110110 \times 2^{678},$$

whose logarithm is

$$\log t = \overbrace{111010110.01000111100111101011101001111100100101110001}^{53 \text{ bits}} \underbrace{000000000000000000000000 \cdots 000000000000000000000000}_{65 \text{ zeros}} 1110 \ldots$$

Algorithm 2.2: Subtractive version of L-algorithm [101]

Input: $a, b, d_0 \in \mathbb{R}$ and $N \in \mathbb{N}^*$
Output: Returns the first integer $r \in [\![0, N[\![$ such that $(b - ra) \bmod 1 < d_0$ if one such r exists, else returns an integer greater than or equal to N

```
x ← a mod 1; y ← 1 − x; d ← b mod 1; u ← 1; v ← 1; r ← 0
if d < d_0 then return 0
while true do
    if d < x then
        while x < y do
            if u + v ⩾ N then return N
            y ← y − x; u ← u + v
        end
        if u + v ⩾ N then return N
        x ← x − y
        if d ⩾ x then r ← r + v
        v ← v + u
    else
        d ← d − x
        if d < d_0 then return r + u
        while y < x do
            if u + v ⩾ N then return N
            x ← x − y; v ← v + u
        end
        if u + v ⩾ N then return N
        y ← y − x
        if d < x then r ← r + u
        u ← u + v
    end
end
```

This is a worst case for *directed roundings* on format $\mathbb{F}_{53}$ = binary64, so that

$$\mathrm{HNR}'(\log, \mathbb{F}_{53}, \square) = 53 + 65 = 118.$$

As regards *rounding-to-nearest,* the worst-case of $x \mapsto 2^x$ for binary64 is:

$$v = 1.1110010001011001011001010010011010111111100101001101 \times 2^{-10},$$

whose radix-2 exponential is

$$2^v = \overbrace{1.00000000010100111111110000101110110000101011010100 11}^{53 \text{ bits}}$$
$$0\underbrace{1111111111111111111111111\cdots1111111111111111111111111}_{59 \text{ ones}}0100\ldots,$$

so that

$$\mathrm{HNR}'(x \mapsto 2^x, \mathbb{F}_{53}, \mathrm{RN}) = 53 + 1 + 59 = 113.$$

Remark 2.4 (Complexity of L-algorithm). The Lefèvre algorithm has a complexity of $O\left(2^{\frac{2p}{3}+o(p)}\right)$, where p denotes the precision of the targeted *binary* FP format, so that it is tractable for the binary64 format ($2^{\frac{2\times53}{3}} \approx 4 \times 10^{10}$), and near the limits of current processors for Intel's extended-binary64 format ($2^{\frac{2\times64}{3}} \approx 7 \times 10^{12}$) [160].

2.5.3 The SLZ Algorithm

The Stehlé–Lefèvre–Zimmermann (SLZ) algorithm uses higher degree polynomial approximations, and solves the TMD for those approximations by following an approach inspired by Coppersmith's method [40, 39, 41]. Using larger degree polynomials allows one to use larger, hence fewer, intervals [160, 158], giving a better asymptotic complexity, though it has not been used until then in large-scale computations. A variant of this algorithm solves the Approximate TMD, and can, under a reasonable heuristic assumption, be made polynomial for $q = O(p^2)$, though the practical meaning of this is still to be understood.

Roughly speaking, the SLZ algorithm is based on the bivariate, modular version of Coppersmith's technique and works as follows:

1. Focus on the (Approximate-)TMD problem over small sub-intervals (typically a sub-interval must be tight enough to ensure that for a given $b \in \mathbb{Z}$, the exponent of $\beta^b \times f(x)$ is zero over the interval, and at the same time for Coppersmith's technique to be applicable), which amounts to finding the values of $x \in \mathbb{Z}$ that are solutions of:

$$|x| \leqslant A \quad \wedge \quad \left|\beta^b f\left(x_0 + \frac{x}{\beta^a}\right) \bmod \beta^{1-p}\right| \leqslant \frac{1}{\beta^q}, \tag{2.36}$$

for fixed integers p, q, a, b, A, and $x_0 \in \mathbb{Q}$,
which amounts to saying that there exists a value $z \in \mathbb{R}$ such that

$$|x| \leqslant A \quad \wedge \quad |z| \leqslant \min\left(\tfrac{\beta^{1-p}}{2}, \tfrac{1}{\beta^q}\right) \quad \wedge \quad \beta^b f\left(x_0 + \tfrac{x}{\beta^a}\right) \equiv z \pmod{\beta^{1-p}}; \tag{2.37}$$

2. Replace f with a polynomial approximation[3] $F \in \mathbb{Q}[X]$ around x_0, so that the initial problem (2.37) implies the existence of a value $z' \in \mathbb{R}$ satisfying

$$|x| \leqslant A \quad \wedge \quad |z'| \leqslant \frac{1}{\beta^q} + \epsilon \quad \wedge \quad \beta^b F\left(\frac{x}{\beta^a}\right) \equiv z' \pmod{\beta^{1-p}}, \tag{2.38}$$

where $\epsilon > 0$ takes the approximation error into account (ϵ will typically be a rational chosen such that $\epsilon \lesssim \frac{1}{\beta^q}$);

3. Clear the denominators[4] in Equation (2.38), so that the problem can be rewritten as finding all the solutions $(x, y) \in \mathbb{Z}^2$ of an equation of the form

$$|x| \leqslant A \quad \wedge \quad |y| \leqslant B \quad \wedge \quad Q(x, y) \equiv 0 \pmod{M}, \tag{2.39}$$

for given $Q \in \mathbb{Z}[X, Y]$ and $M, A, B \in \mathbb{Z}$. We have thus reduced the problem (2.36) of finding the *small real values* of a *univariate function* modulo a rational number, to the one of finding the *small integral roots* of a *bivariate polynomial* modulo an integer;

4. Compute, with the help of the LLL lattice base reduction algorithm [104], two linear combinations v_1 and v_2 with small integer coefficients of the family of polynomials

$$Q_{i,j}(X, Y) = M^{\alpha - i} Q(X, Y)^i Y^j \quad \text{for } (i, j) \in [\![0, \alpha]\!] \times \mathbb{N},$$

where the Coppersmith parameter $\alpha \in \mathbb{N}^*$ is chosen to maximize the reachable bounds A and B. The polynomials v_1 and v_2 with small coefficients are hard to find (this is a call to the LLL algorithm) but easy to check, given the coefficients of the linear combination. Thus, suppose we have found such polynomials that are sufficiently small compared with M^α, in the sense that they satisfy

$$\forall x, y \in \mathbb{Z}, \quad |x| \leqslant A \ \wedge \ |y| \leqslant B \implies \begin{cases} |v_1(x, y)| < M^\alpha \\ |v_2(x, y)| < M^\alpha \end{cases} \tag{2.40}$$

First, we notice that any solution (x, y) of (2.39) will be a modular root of each $Q_{i,j}$ modulo M^α, hence a modular root of v_1 and v_2 modulo M^α. Then, combined with (2.40), this implies that (x, y) is an integral root of both v_1 and v_2;

5. Find the solutions of the polynomial system

$$\begin{cases} v_1(x, y) = 0 \\ v_2(x, y) = 0, \end{cases} \tag{2.41}$$

subject to $|x| \leqslant A$ and $|y| \leqslant B$, over the integers.

Remark 2.5 (Complexity of SLZ in the typical case). If we denote by p the precision of the targeted FP format, and if $2p$ is the value chosen for the parameter q that appears in Definition 2.17, the SLZ algorithm runs with an asymptotic complexity of $O\big(2^{\frac{p}{2}+o(p)}\big)$ [159, Corollary 4].

[3] Choosing in practice multiple-precision *FP numbers* for the coefficients of F.

[4] Noticing beforehand that $z' \in \mathbb{Q}$, and that $u \equiv v \pmod{w} \Rightarrow N \times u \equiv N \times v \pmod{N \times w}$.

Remark 2.6 (Reasoning by implication). Note by the way that the overall algorithm proceeds by implication (especially due to steps 2 and 4), which means that at the very end, the pairs (x, y) obtained at the root-finding step 5 are candidates to be checked against Equation (2.36). Trusting the results computed by SLZ thus amounts to saying that all the actual solutions of the considered instance of the TMD problem are indeed among this list of candidate solutions, so that *we have not forgotten any hard-to-round case*.

Chapter 3

Formal Methods and Interactive Theorem Proving

3.1 Overview of Formal Methods

Formal methods are used in areas where errors can cause loss of life or significant financial damage, as well as in areas where common misunderstandings can falsify key assumptions. In such areas like critical software and hardware systems, formal methods allow one to ensure a high-level of safety and security, even when an exhaustive testing would be intractable. There exist several techniques, all being based on specific mathematical methodologies, for instance:

- abstract interpretation,
- model checking,
- SAT/SMT solvers (based on satisfiability),
- temporal logic,
- theorem proving.

Most of them are automated techniques, while in the sequel we will mainly focus on formal theorem proving, which relies on tools that are strongly related to human-directed proofs.

A *theorem prover* is a piece of software that allows one to (interactively) develop and check mathematical proofs. Basically, we can write formal specifications, programs, and proofs that the considered programs meet their specifications.

There exist various criteria allowing one to classify the various existing theorem provers. We will quote five typical ones below, and the reader will be able to get more details in [164]:

the kind of underlying logic that may be a *first-order* or *higher-order logic*, *constructive* or not, with or without *dependent types*, etc.;

the De Bruijn criterion asserts the existence of a proof kernel, devoted to the "post-verification" of each generated proof object;

the Poincaré principle tells whether the system allows one to discharge some proofs by a mere computation: this principle originated in Poincaré's statement that "a reasoning proving that $2 + 2 = 4$ is not a proof in the strict sense, it is a verification" [142, chap. I];

the kind of proof language that may be a declarative language (cf. Mizar [131]), a language of terms (cf. Agda [136]), or a language of tactics (cf. PVS [139], Coq [10]) inspired by the LCF system [121];

the degree of automation which has a strong impact on the ease of developing formal proofs in the system, but may also influence the readability of these proofs.

Note also that the expression "theorem prover" admits synonyms such as "proof assistant" and "proof checker", the former insisting somewhat on the elaboration on the formal proof, and the latter on the verification of the so-obtained formal proof by the kernel.

Using these tools naturally raises the issue of the soundness of the proof checker. As explained in [143], believing a machine-checked proof can be split in two subproblems:

- "deciding whether the putative formal proof is really a derivation in the given formal system (a formal question)";
- and "deciding if what it proves really has the informal meaning claimed for it (an informal question)".

To some extent, this two-fold approach can be compared to the *verification and validation* (V&V) concept that is typical in formal methods: the *verification* step that aims at ensuring the product meets the *specifications* (roughly speaking, "Are we building the thing right?"), and the *validation* step aims at ensuring these specifications meets the user's needs (roughly speaking, "Are we building the right thing?").

3.2 The Coq Proof Assistant

Coq [10, 81, 162] is an interactive formal proof assistant, based on an expressive formal language called the *Calculus of Inductive Constructions* (CIC), which is a constructive, higher-order logic with dependent types.

The Coq system comes with a relatively small proof kernel that may fit in with independent verification. Specifically, the soundness of any (axiom-free) proof that we may formalize in Coq only relies on the consistency of Coq's logic and *the correctness of the kernel,* which we may just as well re-write in another language, and compile on various machines and platforms.

Finally, we would like to highlight the fact that the use of a proof assistant such as Coq guarantees that all proofs of a given theory are *correct,* provided the definitions involved in its theorems (including hypotheses) are *relevant.* As mentioned in the remark on V&V in the previous section, this issue is mostly an informal question. Yet this is something well *surveyable,* which consists most of the time to check that the formal definitions indeed correspond to the usual mathematical ones. On the other hand, the burden of the verification, namely

the whole of the proof details involved in the derivation, is fully handled by the tool. In other words, as lengthy as the proof might be, the confidence we have in such a mechanized proof does not depend on its size.

3.2.1 Overview of the Tools and Languages Involved in Coq

The Coq proof assistant comes with several command line programs which are implemented in OCaml, including:

- `coqtop`, the interactive toplevel of Coq;
- `coqc`, the Coq batch compiler;
- `coq_makefile`, to ease the compilation of complex Coq developments. Typically, we can compile a `coq_makefile`-based project by running:

```
$ coq_makefile -f Make -o Makefile && make
```

In Coq both specifications and proofs are gathered in a single file, in a human-readable way. This is accomplished thanks to the so-called Vernacular language[1] that provides commands such as `Definition` and `Lemma`, allowing the user to structure its Coq script.

We can save such Coq scripts as vernacular files (i.e., ASCII files with extension '`.v`') using any basic text editor. Yet some GUIs are also available for editing '`.v`' files, including CoqIDE and ProofGeneral which is based on the customizable text editor Emacs.

Then, a Coq *library* will usually consist of several *theories* (each of them being stored in a vernacular file). A given theory can import other theories, e.g. to load some given prerequisites contained in `foo`, we can invoke the command `Require Import foo`. This assumes that the theory `foo` has been successfully compiled, namely that a compiled file `foo.vo` has been produced. Note that when combining many theories in a given Coq development, the dependencies resulting from the `Require` commands have to form a *directed acyclic graph* (DAG).

The specification language of Coq, called Gallina, is a mathematical high-level language that offers the expressiveness of a strongly-typed functional programming language, directly wired to the higher-order logic CIC. We will describe most of features of this language in Section 3.2.2.

As regards the proofs, Coq provides a language of tactics that we will briefly present in Section 3.2.3. Note that the system allows one to define custom tactics in order to increase automation.

Moreover, CIC provides a primitive notion of computation that we will explain in more detail in Section 3.2.4.

Section 3.2.5 will be devoted to a general survey of a number of technical notions that are typical when developing formalizations in Coq.

Then, in order to position our Coq developments carried out in this thesis, we will recall in Section 3.2.6 the different approaches that are possible when dealing with algorithms in a formal setting, while Section 3.2.7 will be devoted

[1] It is a bit like a natural language since sentences begin with a capital letter and end with a dot.

to a brief presentation of the existing Coq libraries that are related to our formal developments.

Note that we will use the same syntax highlighting as the one of ProofGeneral-Emacs, which can be summarized as follows:

```
Command (e.g., Lemma)
keyword (e.g., forall)
tactic (e.g., apply)
tactical (e.g., repeat)
terminator (e.g., done) (borrowing the terminology from [64])
(* and comments, with OCaml's syntax (* possibly nested *) *)
```

3.2.2 Overview of the Gallina Specification Language

In Coq, *all* objects have a type. Note that this is also the case for the dummy variable of all quantifiers. If e.g., expressions such as "$\forall x, P$" are valid in set-theory, in a type-theory setting the x variable must belong to a type nevertheless: we should say instead "for all x belonging to T, P holds". The expression "x belonging to T" is written "x `:` T". One can also say "x is an *inhabitant* of type T", or simply "x has type T". In this case, we will say that x is a (well-formed) *term*.

Moreover, given two types `T` and `U`, we can build the *arrow-type* "`T -> U`," which corresponds to the type of all the functions[2] from `T` to `U`. Using the keyword '`fun`' we can build so-called λ-abstractions, that is to say anonymous functions. For instance the function "`fun t : T => t`" will have type "`T -> T`," and the multiple-arguments function "`fun (t : T) (u : U) => t`," equivalent to "`fun t : T => fun u : U => t`," that is to say $t \in T \mapsto (u \in U \mapsto t)$ in a mathematical fashion, will have type "`T -> (U -> T)`," which is usually abbreviated as "`T -> U -> T`" by agreeing that the Coq arrow is right-associative.

We can also apply functions to one or more arguments. For instance, assuming that `f` has type "`T -> U -> V`," and that we have two terms "`x : T`" and "`y : U`," we can just write "`f x`" to apply `f` to `x`, and finally write "`(f x) y`" to get an object of type `V`. Note that this can be shortened to "`f x y`," as the Coq syntax assumes any such succession of applications is left-associative. Note also that as this example points out, one usually prefers to define multiple-arguments functions in Coq in a *curried* way, that is to say with successive abstractions, e.g. "`fun (t : T) (u : U) => v : V`" leading to a type "`T -> U -> V`," rather than in an *uncurried* way, i.e. providing an inhabitant of "`T * U -> V`." The latter is quite natural in our mathematical habits (viz., we tend to write $f(x, y)$ rather than $f(x)(y)$), but in Coq it would lead to an extra operation, namely the destruction of the considered elements of type "`T * U`" for extracting their components.

To display the *type* of a given term `t`, one can use the command "`Check t`," which will raise an error when the expression `t` is not well-formed. For instance, we can display the type of the addition of natural numbers:

```
Check plus.
    ⇝ plus : nat -> nat -> nat
```

[2] Note that all functions in Coq are total. Still, we will see some techniques to cope with partial functions later on in Section 3.2.5.

and we can also partially evaluate this function, which gives type:

```
Check (plus 3).
    ⇝ plus 3 : nat -> nat
```

Furthermore, a specificity of the logic of Coq is that types are recognized by the same syntactic class as terms, that is to say, *types* must also be *well-form terms*. But then, what will be their type? If we give a try to interactively guess it, we will get for instance:

```
Check 3.
    ⇝ 3 : nat
Check nat.
    ⇝ nat : Set
Check Set.
    ⇝ Set : Type
Check Type.
    ⇝ Type : Type
```

Let us briefly explain this output. In the CIC, the type of a type is called a *sort*, which will always be `Set`, `Type`, or `Prop` — the sort of *propositions*. `Set` and `Prop` have type `Type`, but the keyword `Type` actually refers to a sort "`Type`(i)" taken in an infinite hierarchy of sorts $\{\texttt{Type}(i) \mid i \in \mathbb{N}\}$ called *universe*. Thus strictly speaking, `Type` is not its own type, else it would lead to an inconsistency [83]. But in practice the level i in the hierarchy is handled transparently to the user. To be more precise, we have "`Type`(i) : `Type`(j)" if and only if $i < j$, yet this will be displayed "`Type : Type`" by the Coq pretty-printer.

As regards the definition of logical propositions, they are understood by the Coq system in the perspective of the so-called Curry–Howard isomorphism, also known as the propositions-as-types correspondence. This means that in CIC, a *proposition* "`G : Prop`" should be considered as a type that gathers terms representing proofs of `G`. In other words, saying that "`g` is a *proof* of *formula* `G`" amounts to saying that "`g` is a *program* of *type* `G`".

A key notion related to propositions is implication. In a mathematical context, especially in pen-and-paper proofs, we usually write it with a double-arrow (e.g., $A \Rightarrow B$). Yet in Gallina, we just write it with a single-arrow (viz., "`A -> B`"), which is fully legitimate in the context of the Curry–Howard correspondence : a proof "`p : A -> B`" will stand for a function that transforms an *arbitrary* proof of `A` into a proof of `B`.

Note that as from Coq 8.0, Gallina's syntax uses another kind of arrow, namely the '`=>`' involved in λ-abstractions such as "`fun t : T => t`", which is really required for syntactic reasons. (For instance, if we had just one kind of arrow, we would not be able to distinguish between "`fun s : T -> U => V`" and "`fun t : T => U -> V`").

Let us give a simple example of λ-term to illustrate the previously mentioned notions: if `G` is the proposition "`forall A : Prop, A -> A`," a Coq proof of `G` will be, for example, the term "`fun (A : Prop) (a : A) => a`," which is indeed a function of type `G`.

Note that a universal quantification '`forall`' naturally appears in the example of proposition mentioned above, but this construction is not specific to propositional logic. The syntax "`forall t : T, U t`" actually corresponds to what we call a *product-type* (or Π-type). We can notice this is very similar to the

notion of the product of a family of sets $(U_i)_{i \in I}$ indexed by a given set I: a function that is member of this product will be a function mapping every $i \in I$ to an element in U_i (to be compared to a Coq function of type "`forall i : I, U_i`").

Note that Gallina's '`forall`' quantifier is not defined but primitive, as well as the arrow '`->`' which actually corresponds to non-dependent product. To be more precise, when the term `U` involved in "`forall t : T, U`" does not depend on the value of "`t : T`," we can replace `t` with a wildcard, namely write "`forall _ : T, U`," which is automatically shortened to "`T -> U`" by the pretty-printer.

Note also that the '`forall`' quantifier allows both first-order and *higher-order* quantification. For dependent products, this means roughly that the type of the bound variable may be just as well an arrow-type (or even a product-type). Consequently, within Coq we can smoothly consider higher-order functions such as

```
Coq < Check List.map.
map
     : forall A B : Type, (A -> B) -> list A -> list B

(* as well as *)

Coq < Check List.Forall.
Forall
     : forall A : Type, (A -> Prop) -> list A -> Prop
```

In addition to the `Check` command that we have used several times up to now, note that a vernacular command `Print` is available to display the λ-term of any defined identifier, including the proof term of any successfully-proved theorem.

Inductive Types

Now we focus on the concept of inductive type, which is a very expressive feature of the Coq system that allows one to easily define complex types and provides standard ways to manipulate them.

The idea conveyed by the concept of inductive type is very natural, since similar to how we usually proceed in (object-oriented) programming languages when defining a data structure: roughly speaking, we start by specifying the different ways we can "build" the desired datatypes (this is the role of so-called *constructors*), then we specify how we can "use" such objects (e.g., for inspecting the content of a previously built object). The CIC formal language provides a unified way to declare such objects, and Gallina uses the following general syntax for defining an inductive type `ident`:

```
Inductive ident binder_1 ... binder_k : type :=
  | ident_1 : type_1
  | ...
  | ident_n : type_n .
```

First, such a definition creates a type `ident` and the identifiers `ident`$_1$... `ident`$_n$, which are called *the constructors of inductive type* `ident`. At the same time, it generates some induction principles `ident_ind`, `ident_rec` and `ident_rect`

(resp. for sorts `Prop`, `Set` and `Type`), which describe intuitively the fact that `ident` is the smallest object satisfying the clauses $\texttt{ident}_1 \ldots \texttt{ident}_n$. These induction principles are also called *elimination principles*, *destructors* or *recursors*.

Most of data-types can be defined this way, including the type of Booleans, Peano natural integers, and polymorphic lists:

```
Inductive bool : Set :=
  | true : bool
  | false : bool.

Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.

Inductive list (A : Type) : Type :=
  | nil : list A
  | cons : A -> list A -> list A.
```

Note that while `bool` can just be viewed as an enumerated type, both types `nat` and `list` are recursive (the argument `(A : Type)` in parentheses, written just below the typing colon of the inductive, is called a *parameter*).

We will even see *infra* that almost all logical connectors can be defined using an inductive definition.

Furthermore, if the concept of "record" is a wide-spread way to define simple datatypes in usual programming languages, it is available in Coq through the commands `Record` and `Structure`, which are synonymous: they define the desired type in the form of an inductive type with a single constructor, along with some handy functions to access the different fields of the record (called *accessors* or *projections*).

For instance, the following invocation defines complex numbers as pairs of two real numbers:

```
Record C : Set := Cplx { Re : R ; Im : R }.
```

This is roughly equivalent to process the following commands:

```
Inductive C : Set :=
  | Cplx : R -> R -> C.
Definition Re : C -> R :=
  fun z : C => let 'Cplx x y := z in x.
Definition Im : C -> R :=
  fun z : C => let 'Cplx x y := z in y.
```

and a dot-notation is also be provided for the specified accessors (here "`z.(Re)`" will be a synonymous of "`Re z`", and similarly for `Im`).

We can actually define more involved `Record`s, for example with some fields depending on the value of the others fields. This includes the case where a field of the record is a proof that ensures the other fields satisfy a given property.

The usual equality in Coq is defined by the inductive predicate

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  | eq_refl : eq A x x.
```

that can be viewed as a family of predicates "being equal to `x`", or also as the smallest relation on `A` that is reflexive. Coq uses the notation "`x = y`" as an abbreviation of "`eq _ x y`," where x and y must have the same type. In general this type may be inferred by the system, hence the wildcard. Otherwise Coq provides an extra notation "`x = y :> A`" for "`eq A x y`." Given the form of the elimination principle of `eq`, this kind of equality is very often called *Leibniz's equality*:

```
Check eq_ind.
    ⇝ forall (A : Type) (x : A) (P : A -> Prop),
         P x -> forall y : A, x = y -> P y
```

We have already seen that universal quantification and implication are primitive notions in Gallina. As regards the other logical connectors, they are all defined in the core library of Coq in the form of an *inductive type*, except the negation and the equivalence (bi-implication) that are not "atomic".

Thus the always-false proposition is defined as

```
Inductive False : Prop := .
```

i.e., an inductive in `Prop` with no constructor, while the always-true proposition is defined as

```
Inductive True : Prop :=
  | I : True.
```

Then the logical negation is defined as

```
Definition not (A : Prop) := A -> False.
```

along with the unary notation

```
Notation "~ x" := (not x) : type_scope.
```

Then the conjunction is defined with its infix notation in one shot:

```
Inductive and (A B : Prop) : Prop :=
  | conj : A -> B -> A /\ B
where "A /\ B" := (and A B) : type_scope.
```

and similarly for the disjunction:

```
Inductive or (A B : Prop) : Prop :=
  | or_introl : A -> A \/ B
  | or_intror : B -> A \/ B
where "A \/ B" := (or A B) : type_scope.
```

Finally, the "if and only if" connector is defined as

```
Definition iff (A B : Prop) := (A -> B) /\ (B -> A).
```

As regards existential quantification, an inductive type

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  | ex_intro : forall x : A, P x -> ex P.
```

is provided, along with an intuitive notation "`exists x : A , p`" that stands for "`ex (fun x : A => p)`".

Before entering the upcoming subsection, let us elaborate on one of the inductive predicate mentioned *supra*, namely the disjunction. It has two constructors `or_introl` (resp. `or_intror`) that are intended to build an object of type "`A \/ B`" from a proof of `A` (resp. `B`). Consequently, each constructor correspond to what is called an "introductory rule" in natural deduction logic, namely the operation that consists of *introducing* a new connector (here it is '`\/`') using the required hypotheses. The opposite operation, called "elimination rule", consists of *using* an available assumption written with the considered connector; roughly speaking, it is made possible in Coq by means of the *elimination principles* that are generated by Coq at the time the inductive type is defined. For example, we can ask the Coq *toplevel* to display the *elimination principle* of the disjunction:

```
Check or_ind.
    ⇝ or_ind : forall A B P : Prop,
         (A -> P) -> (B -> P) -> A \/ B -> P
```

This means that for proving a given proposition `P`, if we have a proof of "`A \/ B`", it suffices to prove that "`A -> P`" and "`B -> P`" hold.

As a matter of fact, for *proving* that a disjunction "`A \/ B`" holds, we need to say explicitly which of `A` and `B` holds (and provide a proof for it), which will be accomplished thanks to the Coq *tactics* `left` and `right`, w.r.t. the constructors `or_introl` and `or_intror`. One such proof of disjunction might be tricky, even if `B` is replaced with `~A`, since the underlying logic of Coq is *intuitionistic*. In other words, we do not have by default the *axiom of excluded-middle*

```
Axiom classic : forall A : Prop, A \/ ~A.
```

which is typical in *classical logic*. However, this axiom can be added to the context just by load the appropriate standard library

```
Require Export Classical.
```

(Concerning the use of classical logic in Coq, we will give some details on consistency issues later on in Section 3.2.7.)

Interestingly enough, we cannot use such a logical disjunction to build "*informative data*" in `Set` or `Type`, since there is no elimination principle "`or_rec`" nor "`or_rect`". This is due to the design of the *CIC* that enforces the fact that objects in sort `Prop` have no informative content, thereby their content should not been involved in building informative objects in `Set`/`Type`. Consequently, we can "destruct" objects in sort `Prop` only to build objects in `Prop` again.

Nevertheless, the Coq core library provides some inductive types in sort `Type` for this purpose. They are very similar to the ones in sort `Prop` that we mentioned *supra*. Consider for example:

```
Inductive sumbool (A B : Prop) : Set :=
  | left : A -> {A} + {B}
  | right : B -> {A} + {B}
where "{ A } + { B }" := (sumbool A B) : type_scope.
```

Like "`or`", this 2-constructors inductive definition gathers a proof of either `A` or `B`, yet it behaves just like a Boolean type. To sum up, we can build algorithms in `Set`/`Type` that rely on the content of a given "`C : {A} + {B}`."

Pattern-Matching

A major feature of Gallina is the capability of inspecting the content of an inductive object to perform a definition by case-analysis. This is accomplished with the help of so-called *pattern-matching* expressions.

Basically, one such expression has the form

```
match term_0 with
  | patt_1 => term_1
  | ...
  | patt_n => term_n
end
```

with the following constraints:

- the `term`$_0$ belongs to an inductive type `I`;
- the pattern `patt`$_i$ in each branch is a term of the form "`c x y ...`" where `c` is a constructor of `I` and `x`, `y`, ... are either a wildcard or an identifier, which can then occur in `term`$_i$;
- the patterns are non-redundant and matches each constructor of the inductive type `I`.

For instance, the following command defines the predecessor function for Peano natural numbers:

```
Definition pred (n : nat) : nat :=
  match n with
    | O => O
    | S n' => n'
  end.
```

while the function that returns the tail of a list may be defined by:

```
Definition tail (A : Type) (l : list A) :=
  match l with
    | nil => nil
    | cons _ l' => l'
  end.
```

Note that in the *pattern-matching* expressions presented above, the result type of the overall expression does not depend on the matched term and can be inferred from the common type of the branches. If it is not the case, Gallina provides a more general syntax for dependent pattern-matching, with some "`as... in... return...`" annotations that are written just before the `with` keyword [162, Section 1.2.13].

Pattern-Matching for Singleton and Boolean Types. Gallina provides specific notations to ease pattern-matching on types that have one or two constructors.

To be more precise, if `T` is an inductive type with a single constructor `BuildT` taking n arguments, the expression

```
match term with
  | BuildT ident_1 ... ident_n => term'
end
```

can be equivalently written

```
let 'BuildT ident_1 ... ident_n := term in term'
```

Such inductive types with a single constructor are sometimes called singleton types. A typical kind of singleton types are Gallina's `Record`s, which we mentioned above.

On the other hand, an inductive type with exactly two constructors may behave like a Boolean type if it is involved in a pattern-matching expression where the arguments of both constructors are ignored. In such a case an "`if...then...else...`" syntax is available. More precisely, if `term` has type `B` that is an inductive type with two constructors $\texttt{C}_1$ and $\texttt{C}_1$,

```
match term with
| C_1 _ ... _ => term_1
| C_2 _ ... _ => term_2
end
```

can be shortened to

```
if term then term_1 else term_2
```

In particular the pretty-printer will use this syntax for all the types that appear in `Print Table Printing If`, including `bool` and `sumbool`.

Recursive Functions. In Gallina we can define recursive functions in different ways. The most common one relies on the concept of *fixpoint*. When defining such an object, Coq enforces that the considered function has a "structural decreasing argument," that is an argument whose type is an inductive one and whose "size" decreases at each recursive call. More formally, its new value must be a sub-term of the previous value. We can specify which argument has this role using an explicit annotation,[3] such as `{struct arg}`. For instance, the addition of two (Peano) natural numbers can be defined as follows:

```
Fixpoint plus (n m : nat) {struct n} : nat :=
  match n with
    | O => m
    | S p => S (p + m)
  end
where "n + m" := (plus n m) : nat_scope.
```

It should be noted that the structural decreasing condition mentioned just now is required to ensure the consistency of the system. Indeed, if we could define the following function "`wrong : nat -> nat`," we could derive a proof that "`wrong 0 = 1 + wrong 0`" and its contrary. This would implies that `False` has an inhabitant,[4] and thereby that every formula holds[5].

[3] otherwise Coq ($\geqslant$ 8.2) will choose the first argument that satisfies the condition.
[4] since `P /\ ~P` is by definition equivalent to `P /\ (P -> False)`, which implies `False`.
[5] cf. e.g., the elimination principle of `False`.

```
Fixpoint wrong (n : nat) {struct n} : nat :=
  1 + wrong n.
```

More straightforwardly, we could give an inhabitant to any type `T` by defining:

```
Fixpoint wrong' (n : nat) {struct n} : T :=
  wrong' n.
```

Both pitfalls are avoided thanks to the the structural decreasing condition.

3.2.3 The Coq Proof Language

In Coq as in most *interactive* proof assistants, the proofs are interactively built by the user using a specific proof language. There actually exists two proof languages in Coq, namely `C-zar` (a declarative one *à la* Mizar), and Ltac (language of tactics) [48]. The latter language is by far the most widely used and it is the one we will focus on throughout this document.

Thus each step of a Coq proof is written with the help of so-called tactics. The Coq system comes with many built-in tactics. (Since Coq-v8, they are all written with an initial lower case, as opposed to vernacular commands.) Note that the user may also develop custom tactics using the `Ltac` vernacular. Some tactics take one or more arguments, in general separated with a space. In any case, any tactic invocation must end with a dot. Yet we may combine several tactics in a single proof sentence with the help of so-called *tacticals* (including the concatenation of tactics with '`;`' that we will explain below, as well as the keywords `try`, `repeat`, etc.).

Basically, the development of a Coq proof is a bottom-up process, starting from the conclusion of the theorem to be proved. Then, with the help of tactics, one such proof goal is turned into one or more simpler subgoals, and so on, until we reach trivial subgoals. Note that at each step, the set of subgoals generated using a given tactic consist of *sufficient* conditions to solve the initial subgoal. Finally, we obtain a kind of proof tree, whose nodes can be viewed as intuitionistic sequents, that is, logical objects of the form "$H_1, H_2, \ldots, H_m \vdash C$" where the conclusion C has to be proved under the hypotheses $H_1, \ldots, H_m$ of the context. At the beginning of the proof, there are no hypothesis H_i in the context, but the conclusion C corresponds to the statement of the theorem to be proved. In particular, it should be noted that this "statement to be proved" is a type, and the proof process amounts to interactively building an inhabitant of this type.

To briefly illustrate this proof process, we focus on a toy example, namely proving the "identity principle" saying that "$\forall P,\ P \Rightarrow P$". This leads to the following Coq session:

```
Coq < Lemma id_princ : forall P : Prop, P -> P.
1 subgoal

  ============================
   forall P : Prop, P -> P
```

where the formula under the double-bar is the conclusion to be shown, under the hypothesis located above this double-bar. Currently, there is no such hypothesis, but we can start by introducing some in the context:

```
id_princ < intros P.
1 subgoal

  P : Prop
  ============================
   P -> P
```

Mathematically speaking, this step amounts to saying "Let `P` be a proposition". Now, "let us assume that `P` holds":

```
id_princ < intros H.
1 subgoal

  P : Prop
  H : P
  ============================
   P
```

Then the subgoal is trivial, since we just need to provide an inhabitant of type `P` assuming `H` is of type `P`:

```
id_princ < exact H.
Proof completed.
id_princ < Qed.
id_princ is defined
```

And the proof is finished. (This tautology could have been proved just as well using the `tauto` tactic.) By curiosity, we can ask Coq to display the proof term that has been generated:

```
Coq < Print id_princ.
id_princ = fun (P : Prop) (H : P) => H
     : forall P : Prop, P -> P
```

and we can see that this proof term is nothing but an identity function, and that it indeed has the type "`forall P : Prop, P -> P`".

Thus, the use of tactics not only allows one to guide the building of the interactive proof tree, but also results in the definition of a proof term, whose type can then be checked. This observation is strongly linked to the Curry–Howard correspondence that we mention earlier in Section 3.2.2: a Coq proof is nothing but a *type judgement* of the form

$$E \vdash \texttt{proof} : \texttt{T}$$

where `T` is the statement of the theorem inside the environment E.

There exists some major categories of tactics, including:

- tactics for *minimal propositional logic* (including negation): `intros`, and `apply`, `exact`, `assumption`;
- tactics to *introduce usual inductive types*:
  `reflexivity` to prove the trivial equality "`x = x`" for a given `x`;
  `left` and `right` for the "`or`" logical connector;
  `split` for the conjunction "`and`";
  `exists` for providing a witness to an existential quantification,
  as well as the `constructor` tactic, applicable for an arbitrary inductive;

- tactics to *eliminate inductive hypotheses*:
  `destruct` to perform case-analysis, or `induction` to apply the appropriate elimination principle;
  `rewrite` to use hypotheses involving equality;
  `discriminate` and `injection` to use basic assumptions on CIC inductive types w.r.t. equality (e.g., a constructor is always injective);

- the *conversion tactics*, including `unfold`, `change`, `simpl` and `compute`;

- some key tactics such as `set` and `pose` to bring in a local definition in the proof context, as well as `assert` to locally prove an intermediate lemma, which is often called a *cut* and leads to a ζ*-redex* in the final proof term (i.e., a term of the form "`let tmp := ... in ...`").

Since a tactic such as `split` or `apply` can possibly generate more than a single subgoal, the '`;`' tactical can be useful to shorten the proof script by invoking the same tactic on all the subgoals generated by the first tactic. For example the proof sentence "`split; trivial.`" is equivalent to

```
split. 2:trivial. trivial.
```

In addition to the few tactics presented above, Coq comes with many other tactics that are often composed with more atomic tactics, and may provide automation for a given kind of subgoals. This includes `auto` (for recursive application of selected lemmas), `tauto` (for intuitionistic propositional calculus), `ring` (for equalities on a declared ring or semi-ring), `field` (likewise, for fields), as well as `omega` (for Presburger arithmetic).

3.2.4 Computing Within the Coq Proof Assistant

First of all, let us recall some preliminaries notions and notations.

Bound Variables and Free Variables. Among the variables that occur in a given formula, some are bound to a quantifier or a similar construct ('`forall`', '`exists`', '`fun...=>...`', '`let...in...`', etc.). The others are called *free variables*, and we will write $\mathcal{FV}(E)$ to denote the set of the free variables involved in an expression E.

***alpha*-conversion.** As in usual mathematics, bound variables are sometimes called "dummy" variables, since the meaning of the overall expression does not depend on the name of these variables. Thus, "`forall x : nat, x = x`" and "`forall n : nat, n = n`" represent the same term. The process of renaming a bound variable is called α-conversion. It is handled transparently to the Coq user and it may be required in some situations to avoid some clash with other variables, like in the example described in the upcoming paragraph.

Substitution. If t is a term, the process of substituting all the occurrences of $v \in \mathcal{FV}(t)$ by another term t' leads to a new term that we will write "$t\,[v := t']$." Some α-conversion is required if some t' contains some free variables whose name clash with the one of a bound variable in t. For instance, if we consider the terms $t :=$ "`fun n => a * n + b`" and $t' :=$ "`n + 1`," then writing the term

corresponding to "$t\,[\mathtt{a} := t']$" needs to rename the dummy variable "`n`" into, say, "`x`," and we obtain the term "`fun x => (n + 1) * x + b`". (Without the α-conversion, we would have (wrongly) obtained "`fun n => (n + 1) * n + b`".)

Now let us focus on the rules that govern computation in the Coq proof assistant. Coq comes with a primitive notion of computation called conversion, which is based on a set of elementary transformations (called reductions) on the terms of CIC. We briefly describe below the four kinds of reductions at stake.

***beta*-reduction.** This first kind of reduction has a central role for evaluating functions: it transforms any expression of the form "`(fun x : T => s) t`" (called β-redex) into "$\mathtt{s}\,[\mathtt{x} := \mathtt{t}]$," that is the term `s` where all occurrences of the free variable `x` are replaced with `t`. Moreover, we usually say that "$\mathtt{s}\,[\mathtt{x} := \mathtt{t}]$" is the β-contraction of "`(fun x : T => s) t`" and that "`(fun x : T => s) t`" is the β-expansion of "$\mathtt{s}\,[\mathtt{x} := \mathtt{t}]$." More formally, we will denote this relation between terms modulo β-reduction in the following manner:

$$\mathtt{(fun\ x\ :\ T\ =>\ s)\ t} \rhd_\beta \mathtt{s}\,[\mathtt{x} := \mathtt{t}]\,.$$

***zeta*-reduction.** This reduction consists of transforming terms by removing the *local definitions*, namely the definitions that are expressed using the primitive syntax '`let...in`'. Thus "`let x := t in s`" becomes "$\mathtt{s}\,[\mathtt{x} := \mathtt{t}]$," which we will denote in the following manner:

$$\mathtt{let\ x\ :=\ t\ in\ s} \rhd_\zeta \mathtt{s}\,[\mathtt{x} := \mathtt{t}]\,.$$

***delta*-reduction.** Since Gallina allows one to define new constants (using the commands `Definition`, `Lemma`, or their synonyms), a specific reduction allows one to replace the constant with its value. This δ-reduction relies on the notion of environment (also called global context) E and context Γ, which gather the definitions added to the system, contrary to the ζ-reduction that only deals with local definitions inserted in the CIC terms themselves. (The context Γ can be understood as the set of the definitions handled by the sectioning mechanism of Coq: we will present this mechanism in Section 3.2.5.) To sum up, if the identifier `d` corresponding to a stored definition $(\mathtt{d} := \mathtt{t{:}T}) \in E \cup \Gamma$ occurs in a term `s`, then the δ-reduction will transform `s` into "$\mathtt{s}\,[\mathtt{d} := \mathtt{t}]$," which can be written as follows:

$$E, \Gamma \quad \vdash \quad \mathtt{s} \rhd_\delta \mathtt{s}\,[\mathtt{d} := \mathtt{t}] \qquad\qquad \text{if} \quad (\mathtt{d} := \mathtt{t{:}T}) \in E \cup \Gamma.$$

***iota*-reduction (and *mu*-reduction [24]).** In CIC, two reduction rules related to inductive types are available, both of them being called ι-reduction in Coq's reference manual [162, Section 8.5.1]. Without going into formal details, the first one allows one to reduce a pattern-matching expression when the matched term begins with a constructor and the second one allows one to reduce a recursive function when its decreasing argument is given in constructor form. For instance, we will have:

$$\mathtt{if\ false\ then\ a\ else\ b} \rhd_\iota \mathtt{b}$$

$$\mathtt{plus\ (S\ n)\ (S\ m)} \rhd_\iota \mathtt{S\ (plus\ n\ (S\ m))}.$$

Meta-theoretical Properties of the Conversion

The four kinds of reductions described above can be used at any position inside a CIC term (including inside the body of a function that is partially applied), so that they are often called *strong reductions*. We will write

$$E, \Gamma \vdash \texttt{t} \rhd_{\beta\delta\zeta\iota} \texttt{t'}$$

to say that `t'` is obtained by one step of any of these reductions in the environment E and context Γ, and

$$E, \Gamma \vdash \texttt{t} \rhd^*_{\beta\delta\zeta\iota} \texttt{t'}$$

to say that `t` can be reduced to `t'` in one or more steps. Any such combination of these four CIC reductions fulfills some key properties that are typical in the study of rewriting systems, including:

strong normalization every sequence of reduction steps is finite, that is to say the process of reduction on a CIC term terminates and leads to a term that is not reducible anymore and called the *normal form*. This is accomplished by the set of constraints that should be enforced for an expression to be accepted as a valid CIC term, including the structural decreasing condition mentioned in the previous section. To sum up, in Coq, every computation *terminates*;

confluence for any terms $\texttt{t}, \texttt{u}, \texttt{v}$, if $\texttt{t} \rhd^*_{\beta\delta\zeta\iota} \texttt{u}$ and $\texttt{t} \rhd^*_{\beta\delta\zeta\iota} \texttt{v}$, then there exists a term `w` such that $\texttt{u} \rhd^*_{\beta\delta\zeta\iota} \texttt{w}$ and $\texttt{v} \rhd^*_{\beta\delta\zeta\iota} \texttt{w}$. Generally speaking, this property is sometimes called the *Church-Rosser property*, or the *diamond property*. In the context of Coq, this implies the uniqueness of the normal form, whatever is the chosen evaluation order, and justifies the availability of the "`Eval compute in ...`" vernacular command to compute *the* normal form of a term. (Note that the `compute` reduction tactic is equivalent to `cbv beta delta zeta iota`.)

subject reduction if $\texttt{t} \rhd^*_{\beta\delta\zeta\iota} \texttt{t'}$ and `t` has type `T`, then `t'` also has type `T`.

Definition 3.1 (Convertibility). Two terms `u` and `v` are said convertible if they reduce to a third common term `w`, that is $\texttt{u} \rhd^*_{\beta\delta\zeta\iota} \texttt{w}$ and $\texttt{v} \rhd^*_{\beta\delta\zeta\iota} \texttt{w}$ for a certain term `w`, which we will denote in the following manner:

$$E, \Gamma \quad \vdash \quad \texttt{u} =_{\beta\delta\zeta\iota} \texttt{v}$$

Note that CIC convertibility is decidable, thanks to the properties of strong normalization and confluence described above. Indeed, to check if two terms `u` and `v` are convertible, it *suffices* to compute and compare their normal form.

A related notion is given by the *convertibility rule* of CIC, which asserts that two *convertible types* have the same inhabitants: this typing rule relies on the relation $T \leqslant_{\beta\delta\zeta\iota} T'$ whose definition extends the convertibility equivalence ($=_{\beta\delta\zeta\iota}$) to take universes into account (e.g., we have `Prop` $\leqslant_{\beta\delta\zeta\iota}$ `Set`).

In order to illustrate how these meta-theoretical properties underlie the behavior of the system, we can consider a toy proof of the following result:

Remark 3.1 (Convertibility implies equality). Suppose x and y (having type T and T', respectively) are two convertible terms in a global context E. Then, $x = y$ in the sense of Leibniz-equality.

Proof. x and y are convertible, so by definition there exists one term u such that $x \triangleright^*_{\beta\delta\zeta\iota} u \triangleleft^*_{\beta\delta\zeta\iota} y$. By subject reduction, u and y admit the same type of x, namely T. Consequently, the terms "G `:= eq` T x y" and "H `:= eq` T u u" are well-typed. Then we notice that "h `:= refl_equal` T u" is an inhabitant of H. Since the Coq reductions are strong, we have $G \triangleright^*_{\beta\delta\zeta\iota} H$, hence $G =_{\beta\delta\zeta\iota} H$, so that by the convertibility rule, G and H have the same inhabitants, implying that $E \vdash h : G$, that is to say $x = y$. □

Reduction tactics. We have seen in the present section that the logic of Coq is computational: it is possible to write programs in Coq that can be directly executed within the logic. This is why the result of a computation with a correct algorithm can always be trusted.

Furthermore, it can be noted that three main reduction tactics are available for computing in Coq:

- `compute`, the original interpreter-based implementation of strong reduction in the Coq system. It is a synonym for `cbv beta delta zeta iota` (where `cbv` stands for "call-by-value");
- `vm_compute`, an implementation of call-by-value evaluation that involves a compilation to the *byte-code of a virtual machine* [67];
- `native_compute`, a similar reduction tactic (call-by-value evaluation strategy) which relies on a compilation to OCaml *native-code* [15]: thanks to this progress in the evaluation mechanism, programs in Coq can potentially run as fast as an equivalent version directly written and compiled in OCaml.

In practice, using `vm_compute` instead of `compute` for a given computation yields a speedup of 10x to 100x, while `native_compute` yields a further speedup of 2x to 5x over `vm_compute`. However it should be noted that doing so somewhat increases what is called the *trusted computing base* (TCB). In particular, the correctness of results that are derived using `native_compute` depends upon the correctness of the entire code of the compiler. But as pointed out in [15], "a certified compiler for the target language would certainly be of interest here to reduce the trusted base."

3.2.5 Some Concepts Involved in an Everyday Use of Coq

Type Inference. First of all, a convenient feature of Coq is type inference, which is typically provided by several strongly-typed functional programming languages such as OCaml and Haskell. It allows the user to omit, if possible, some explicit type annotations, without preventing type checking. For instance, to show that this feature does not contradict what we said in the first paragraph of Section 3.2.2, we can consider the following example: a universal quantification such as "`forall m n, plus m n = plus n m`" will be automatically recognized as "`forall m n : nat, plus m n = plus n m`", since the quantified term involves the function "`plus : nat -> nat -> nat`" whose both arguments should be (Peano) natural integers. However, the expression "`forall x, x = x`" will raise "*Error: Cannot infer the type of* x".

For the purpose of type inference, a special syntax '_' is available in Coq: this "*inference wildcard*" can be used everywhere a type (or even a term) is expected, provided it can be inferred from the context. Note that this underscore symbol '_' has another meaning in the place of a bound variable, namely in such a case, it is a "*non-dependent wildcard*" that stands for any dummy variable name that will not be used elsewhere: for example no inference occurs in "`fun _ : nat => 5`" (a constant function), as well as in "`forall _ : bool, Prop`", which is just another syntax for the non-dependent product "`bool -> Prop`".

Implicit Arguments. Furthermore, Coq provides a powerful mechanism relying on so-called *implicit arguments*, which can be viewed as a way to shorten the terms designation by omitting the inference wildcards themselves. For instance, if a given theory begins with the `Set Implicit Arguments` invocation and defines a polymorphic function such as

```
Definition identity (A : Type) (x : A) := x.
```

then the first argument `(A : Type)` will be declared as implicit, that is to say we can specify the argument `x` just after the function name:

```
Eval compute in (identity 5)
    ⇝ = 5
    ⇝ : nat
```

But if ever we wanted to locally disable the implicit arguments feature, we could use the '`@`' prefix at any time, in the following way:

```
Eval compute in (@identity nat 5).
    ⇝ = 5
    ⇝ : nat
```

Moreover, we can disable the hiding of implicit arguments in the pretty-printer output be setting the flag `Set Printing Implicit`:

```
Check (identity 5).
    ⇝ identity 5 : nat
Set Printing Implicit.
Check (identity 5).
    ⇝ @identity nat 5 : nat
Unset Printing Implicit. (* Back to default setting *)
```

The Coq mechanism of implicit arguments actually consider several kinds of implicit arguments, with appropriate vernacular commands that are fully described in [162].

Coq Sections. The Coq mechanism of sections allows the user to organize his formalization in a somewhat declarative way, by enumerating the variables and constants that will then be locally available in the ambient section. Basically, one can start a section with "`Section Foo`" and enumerate the various hypothesis that are required for a given theorem, then state itself in one line only. Then, when closing the section with "`End Foo`," each variable or local definition is "discharged": their identifier is removed from the context while the theorem is universally generalized with respect these local declarations.

```
Section Foo. (* Let's start a new section *)
Variables a b c : nat.
Let b2 := b * b.
Definition D := b2 - 4 * a * c.
Print D.
    ⇝ D = b2 - 4 * a * c : nat
End Foo. (* Discharging at end of section *)
Print D.
    ⇝ D = fun a b c : nat => let b2 := b * b in b2 - 4 * a * c
    ⇝       : nat -> nat -> nat -> nat
```

Table 3.1 gives a brief summary of the declaration commands that can be used inside a `Section`.

Kind of action	Command with a *local* effect	Equivalent *global* command
Declaring variables	`Variable`/`Variables`	`Parameter`/`Parameters`
Assuming hypotheses	`Hypothesis`/`Hypotheses`	`Axiom`/`Axioms`
Defining terms	`Let`	`Definition`/`Lemma`/`Theorem`

Table 3.1 – *Vernacular commands inside a section*

Modules and Functors

Inspired by the module system of OCaml, Coq provides a native mechanism of modules that allows one to group related Coq objects together [35, 34].

A first implicit use of modules is linked to the organization of a Coq development in several files. One such splitting leads to the creation of one module per file, called *library file*. As these files can possibly be organized in sub-directories, Coq recursively translates this organization in a dots-separated path, under a so-called *library root*. For instance, the main library on polymorphic lists from the Coq standard library has *physical name* `.../theories/Lists/List.v`, thereby its *logical name* will be `Coq.Lists.List`:

```
Locate Library List.
    ⇝ Coq.Lists.List is bound to
        file /usr/lib/coq/theories/Lists/List.vo
```

Each library file can contain some (sub)modules, possibly nested. Furthermore, a key feature of the module system relies on the concept of functor: beyond the "modularization" allowed by bare modules, the concept of functor allows one to define "parametrized modules", where the "shape" of the parameters is specified using *modules types*, also known as *signatures*.

Consequently, the Coq modules allow one to perform a kind of separated compilation, where the implementation at a given level just relies on the interface (i.e., the types of the objects considered) for lower levels, the implementation of these interfaces being stored apart. Moreover, one can just as well define different low-level implementations corresponding to the same module type, and switch between them.

To illustrate how the module system works in practice, here is a simple but complete example using the main vernacular commands related to modules:

```
Module Type MT.
Parameter T : Type.
Parameter Inline U : Type.
End MT.

Module F (A : MT). (* Functor *)
Import A.
Definition Rel := T -> U -> Prop.
End F.

Module M <: MT. (* Implementation *)
Definition T := nat.
Definition U := nat.
End M.

Module FM := F M. (* Instantiation *)
Print FM.Rel.
    ⇝ FM.Rel = M.T -> nat -> Prop
    ⇝       : Type
```

In this example, `F` is a functor taking a module `A` of signature `MT`, which is immediately imported, that is, all the members of `A` are accessible in the body of `F` using short names `T` and `U` (instead of `A.T` and `A.U`). Then, `M` is an implementation of the module type `MT`. Thus, thanks to the clause "`<: MT`", Coq verifies at the moment of "`End M`" that all needed members have been defined (and that they have a correct type). E.g., without the line defining `T`, Coq would raise the "*Error: The field T is missing in Top.M.*" Then, we can instantiate the functor `F` with module `M` (and also notice the effect of the `Inline` invocation involved in `MT` on the definition of `FM.Rel`, viz., `M.U` has been automatically expanded to `nat` at functor application).

Qualified Identifiers. If all the constants (axioms, definitions, lemmas) defined in a Coq development are given an identifier when they are defined, they have also a more precise identifier, called qualified identifier (`qualid`), which consists of the logical path of its parent module, concatenated with its short identifier (with a dot). For instance, the constant `nat` has qualid `Coq.Init.Datatypes.nat`:

```
Locate nat.
    ⇝ Inductive Coq.Init.Datatypes.nat
```

Consequently, if *ever* one defined another object called `nat` in another module, both "`nat`" could be distinguished using their qualid. (But obviously we could not define two different objects with the same identifier in the same module.)

Modules w.r.t. Sections. Note that a Coq `Section` can be opened inside a module, but no `Module` invocation is allowed inside a `Section`.

Limitations of Modules. First of all, when applying a functor, all its arguments have to be (module) identifiers. Consequently, we cannot perform several functor applications at once. Nevertheless, a general workaround consists of

defining intermediate modules to perform the desired instantiation. For instance, instead of writing "`Module M2 := F2 (F1 M)`," one can just write:

```
Module M1 := F1 M.
Module M2 := F2 M1.
```

Finally, the main limitation of modules is that they are not first-class citizens. In particular, we cannot "quantify over a module type" in a Gallina term, and conversely it is not possible to have a Gallina type as a module parameter. Nevertheless, it is always possible to encapsulate such a Gallina parameter inside a module type to do the job.

Alternatives to Modules. In the current Coq system, there exists two other mechanisms of abstraction that are similar to modules, namely *canonical structures* [150] and *type classes* [155]. Both are first-class citizens: they rely on the concept of `Record` and makes heavy use of implicit arguments, in addition to the high-level routines that are specific to each of them (e.g., `Canonical` for the former, and `Class`, `Instance`, `Context` for the latter).

Syntax Customization

Defining Notations. In order to increase the readability and conciseness of the Gallina expressions involved in a Coq library, one can define some user-friendly notations that acts on both Coq's parser and pretty-printer. We can thus customize their syntax through some vernacular commands such as `Notation`. Here is a typical invocation of this command:

```
Notation "x * y" := (prod x y) : type_scope.
```

This makes the '`*`' sign be parsed and pretty-printed as an infix symbol for the `prod` function applied to two given terms (written `x` and `y` in the string). As regards `type_scope`, we will elaborate on this concept of *scopes* later on in this section. Note that we can disable pretty-printing of notations at any moment using the `Unset Printing Notations` command. We can also disable pretty-printing of a notation once and for all, using the "`only parsing`" modifier at definition time. For example, we could write:

```
Notation "x * y" := (prod x y) (only parsing) : type_scope.
```

Syntactic Equality. The status of Coq notations satisfies the following properties:

- The "body" of each notation is *untyped* at definition time; it will actually be type-checked at the time we use the notation. We sometimes says that the body of such a notation is *syntactically equal* to its short form.
- Two variants of the same Gallina expression written with or without using some notations are indiscernible and lead to the same CIC term. Consequently this syntactic equality is "stronger" than the usual equality, or even than convertibility.

Possible Robustness Issues. Since the identifiers chosen in a given library are more likely to change than the notations themselves, relying on notations rather than on identifiers may improve the robustness of the Coq scripts that use this library. Note that when defining a notation it might be necessary to use the '`@`' symbol and/or inference wildcards to cope with implicit arguments. On the other hand, when using existing notations whose body contains such inference wildcards that cannot be filled at type-checking time, it may be better just to add a type-cast colon after the considered notation, rather than using an explicit term without notation. For example, writing "`[::] : _ nat`" instead of "`@nil nat`".

Parenthesizing Issues. Combining several infix notations in a given Gallina expression often leads to ambiguities. They can be avoided using explicit parentheses, or relying on Coq implicit rules for grouping sub-expressions. There exist two kinds of such rules, namely the *operator precedence* that is taken into account to decide the relative "priority" of *two different operators*, and the *associativity* that specifies the way two or more occurrences of *the same operator* are understood. Each new notation should be assigned a precedence level (between `0` and `100`) and a kind of associativity, either at definition time, using the "`at level ...`" and "`... associativity`" modifiers, or using the `Reserved Notation` command, before the `Notation` command itself. Let us illustrate these concepts with the addition and multiplication of (Peano) natural integers. The vernacular invocations

```
Notation "n + m" :=
  (plus n m) (at level 50, left associativity) : nat_scope.
Notation "n * m" :=
  (mult n m) (at level 40, left associativity) : nat_scope.
```

are equivalent to

```
Reserved Notation "n + m" (at level 50, left associativity).
Reserved Notation "n * m" (at level 40, left associativity).
(* and later on : *)
Notation "n + m" := (plus n m) : nat_scope.
Notation "n * m" := (mult n m) : nat_scope.
```

Since the precedence level for multiplication is lower than the one for addition, it will have priority in an expression such as `1 + 2 * 3`, equivalent to `1 + (2 * 3)`. We can actually notice it by examining the output of a `Check` command such as the following:

```
Check 1 + (2 * 3).
    ⇝ 1 + 2 * 3
    ⇝ : nat
```

while the parentheses are obviously mandatory in an expression such as

```
Check (4 + 5) * 6.
    ⇝ (4 + 5) * 6
    ⇝ : nat
```

and the pretty-printer displays the term accordingly. As regards the associativity rule, the term `2 * 3 * 4` will be parsed as `(2 * 3) * 4` given that '`*`'

is left-associative. Thus if we parse and display at once the pair of the similar expressions `2 * (3 * 4)` and `(2 * 3) * 4`, we will obtain:

```
Check (2 * (3 * 4), (2 * 3) * 4).
    ⇝ (2 * (3 * 4), 2 * 3 * 4)
    ⇝ : nat * nat
```

Overloading Notations. In a formal specification as in mathematics, we would often want to reuse the same symbol for similar operations, even if they "do not live in the sames types". Thus an implementation of some overloading "à la C++" is welcome, but it should at the same time be flexible enough and prevent any ambiguity. In Coq, one such overloading is accomplished through the mechanism of scopes. At definition time, each notation is assigned a scope identifier, such as `nat_scope`, `Z_scope`, etc. Each scope gathers several notations (which must be pairwise different), and we can overload any notation in a given scope by the same notation in another scope. For instance, Table 3.2 gives the most usual interpretations of the symbol '`*`'.

Library required	Constant	Type of the constant	Scope name
(Init)	`prod`	`Type -> Type -> Type`	`type_scope`
(Init)	`mult`	`nat -> nat -> nat`	`nat_scope`
`NArith`	`Nmult`	`N -> N -> N`	`N_scope`
`ZArith`	`Zmult`	`Z -> Z -> Z`	`Z_scope`
`QArith`	`Qmult`	`Q -> Q -> Q`	`Q_scope`
`Reals`	`Rmult`	`R -> R -> R`	`R_scope`

Table 3.2 – *Some interpretations of the product symbol* `"*"`

The `Locate` vernacular allows one to display the body and the scope of notations matching a given string among the loaded libraries, for example:

```
Locate "*".
    ⇝ "x * y" := prod x y  : type_scope
    ⇝ "n * m" := mult n m  : nat_scope
    ⇝       (default interpretation)
Require Import ZArith.
Locate "*".
    ⇝ "x * y" := prod x y  : type_scope
    ⇝ "x * y" := Pmult x y : positive_scope
    ⇝ "n * m" := mult n m  : nat_scope
    ⇝       (default interpretation)
    ⇝ "x * y" := Zmult x y : Z_scope
    ⇝ "x * y" := Nmult x y : N_scope
```

Then, the "`Open Scope` ..." vernacular allows one to choose the ambient default scope, while a general syntax `(term)%key` is provided to locally change the default scope, where `key` stands for the *delimiting key* bound to the desired scope. For instance, thanks to the command "`Delimit Scope Z_scope with Z`," the command "`Check (1 + 2 * 3)%Z`" is roughly equivalent to processing the following two commands:

```
Open Scope Z_scope.
Check 1 + 2 * 3.
    ⇝ 1 + 2 * 3
    ⇝ : Z
```

Another useful command is available to automatically interpret functions arguments of a given type with the relevant scope, in order to omit superfluous groupings `(term)%key`, for instance the "`Bind Scope Z_scope with Z`" invocation contained in the `ZArith` library implies that in any function, arguments of type `Z` will be interpreted in `Z_scope` by default.

Finally, note that a handy vernacular command `About` is available to summarize the "status" of most of the concepts mentioned so far, namely the type (like with `Check`), the implicit arguments (like with `Print Implicit`), the arguments scope, and the qualified-identifier (like with `Locate`):

```
About cons.
    ⇝ cons : forall A : Type, A -> list A -> list A
    ⇝       Argument A is implicit
    ⇝       Argument scopes are [type_scope _ _]
    ⇝       Expands to: Constructor Coq.Init.Datatypes.cons
```

Coercions. While the logic of Coq does not provide subtyping in itself, the Coq system implements the concept of `Coercion`. Intuitively, it can be very convenient when we have a function "`i : T -> U`" and we want to "identify" the elements of `T` with their image by `i`. This is useful in situations when an object of type `U` is expected while an argument `x` of type `T` is provided. In this case, declaring `i` as a coercion will automatically consider `(i x)` instead of `x` and the overall term will be well-typed. As regards syntax, declaring the usual injection from `nat` to $\mathbb{Z}$ as a coercion can be written "`Coercion Z_of_nat : nat >-> Z.`"

Extraction

We recall a characteristic of the underlying logic of Coq, related to the Curry-Howard correspondence: a proof is a program. In other words, every proof is a CIC term, which justifies the concept of *extraction* [107]. This mechanism allows one to build programs in ML[6] by extracting them from any Coq function or axiom-free proof that is in the sort `Type` or `Set`. One can extract, for instance, a proof whose conclusion is a `sumbool`, which will lead to a certified decision procedure written in ML.

Definition by Tactics and Opacity

Since there is no visible difference between a Coq proof carried out with tactics and a bare definition of the same term written in Gallina syntax, it is sometimes useful to define a complex function (using dependent types or so) with the help of tactics, such as `refine`. In this case, the very last vernacular used to save the proof plays an important role: if this is `Qed`, the proof is made opaque, i.e., the system will prevent the user from unfolding the proof term, while `Defined` declare the proof as being transparent, just like any Gallina definition.

[6] actually in OCaml, Haskell or Scheme

This concept of opacity has to do with the Coq reduction mechanism, since "unfolding" corresponds to δ-reduction. Intuitively, it makes sense for any "theorem proof", since we can consider the chosen proof path for a theorem (stored in its proof term) is not that important, apart from its existence. This idea corresponds to the so-called proof-irrelevance axiom, which is sometimes assumed for sort `Prop`.

Partial Functions

We recall that for ensuring the logical consistency of the system, all functions defined in Coq have to terminate. Likewise, all functions defined in Coq are total. Yet thanks to the availability of dependent types, it is possible to define partial functions using different strategies.

1. A first kind of partial functions have a type of the form:

   ```
   f : forall x : D, P x -> E
   ```

   where the predicate "`P : D -> Prop`" plays the role of *pre-condition*. If this is probably the most natural way of writing a partial function, in practice it may be tricky to define and use such functions in a given development. This is mainly due to the presence of the proof of "`P x`," which has to be provided for any call of the function `f`. We can define such a function either using dependent pattern-matching (i.e., with a "`match...as...in...return...with...`" Gallina invocation) or with the help of tactics. In the latter case, note that a use of automation tactics such as `auto` or `omega` without caution might lead to a function that is different to the one we would expect. Anyway, the tactic `refine` can take advantage of this situation, by specifying the "skeleton" of the pattern-matching, along with holes for the logical parts of the term, which can be innocuously filled using `auto`, `omega`, and so on.

2. A second strategy consists of defining all the same *a total function,* by choosing a default value "`e0 : E`" when the pre-condition is not verified. This would lead to a function of type

   ```
   f : D -> E
   ```

   while the pre-condition would be kept in the correctness lemmas, like

   ```
   thm : forall x : D, P x -> Q x (f x)
   ```

   denoting by "`Q : D -> E -> Prop`" a given *post-condition*. This approach could seem less natural, yet it turns out to be quite convenient in many situations, and we will often use it in our formal developments.

3. Another strategy consists of defining a total function with an *option-type as a codomain,* namely :

   ```
   f : D -> option E
   ```

   where the polymorphic inductive `option` is defined by:

```
Inductive option (E : Type) : Type :=
  | Some : E -> option E
  | None : option E.
```

Here, the `None` constructor can be seen in some sense as a default value, taken outside `E`.

4. A variant of this strategy consists of using a "*hybrid disjoint sum*" as a codomain, which is very similar to an option-type, but gathers a proof for the "singular" case:

   ```
   f : forall x : D, E + {F x}
   ```

   Intuitively, the "`F x`" may be any helpful proposition corresponding to the case when "`P x`" (the pre-condition of `f`) does not hold. This uses the following definition of `sumor`:

   ```
   Inductive sumor (A : Type) (B : Prop) : Type :=
     | inleft : A -> A + {B}
     | inright : B -> A + {B}
    where "A + { B }" := (sumor A B) : type_scope.
   ```

Note that the 4th strategy is in some sense, stronger than the 3rd one, and so on. Indeed, it is always possible to consider the "projection" of the function defined in a "strong way." For the sake of completeness, we can mention some other inductives that are usually involved in such "*strong specifications*":

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
  | exist : forall x : A, P x -> sig P.
Notation "{ x : A  |  p }" := (sig (fun x : A => p)) : type_scope.

Inductive sigT (A : Type) (P : A -> Type) : Type :=
  |  existT : forall x : A, P x -> sigT P.
Notation "{ x : A  &  p }" := (sigT (fun x : A => p)) : type_scope.

Inductive sum (A B : Type) : Type :=
  | inl : A -> A + B
  | inr : B -> A + B
where "A + B" := (sum A B) : type_scope.
```

The first one, `sig`, is very similar to the inductive `ex` defining to the existence predicate, but contrary to `ex`, `sig` is in sort `Type`. Consequently, as regards extraction, a term of type "`sig E Q`" will successfully be extracted and produce an element of type `E`.

The second one, `sigT`, is a *Sigma*-type (also written Σ-type); it can be useful when considering a nested specification such as, for example:

```
g : forall x : D, {y : E & {z : F | Q x y z}}
```

in order to take into account the specification of a function `g` that would output two values "`y`" and "`z`" for any value `x` in domain `D`, satisfying a given post-condition "`Q : D -> E -> F -> Prop`."

Finally, the last one is the *disjoint sum*, which has to be compared with `sumor` and `sumbool` with respect to the sort of the arguments.

Well-Founded Recursion

As regards the definition of recursive functions and compared to ML languages, Coq enforces their termination through the structural decreasing condition. Yet in some developments, we might need to define functions that do not easily fit this requirement. Nevertheless, Coq allows one to overcome this limitation with the help of so-called well-founded recursions. The idea is to introduce a well-founded relation, with respect to whom the argument of the function to be defined decrease at each recursive call. Then, by definition of a well-founded relation, this process terminates.

In general, this approach relies on the following principle:

```
well_founded_induction_type :
  forall (A : Type) (R : A -> A -> Prop),
  well_founded R ->
  forall P : A -> Type,
  (forall x : A, (forall y : A, R y x -> P y) -> P x) ->
  forall a : A, P a.
```

where `A` is the domain of the function to be defined and "`R : A -> A -> Prop`" is a well-founded relation (e.g., $< : \mathbb{N} \longrightarrow \mathbb{N}$). This leads to a function of type "`forall a : A, P a`," but in practice it may be difficult to reason on the function so-obtained. Nevertheless, it often suffices to consider a strong specification for the function to be defined, and choose the function "`P : A -> Type`" accordingly. Moreover, the term of type

```
forall x : A, (forall y : A, R y x -> P y) -> P x
```

above is the key part of the definition concerning the computation itself, and we can see from this type that a proof of "`R y x`" have to be provided at each recursive call, which may be cumbersome. As a matter of fact, we can here rely on a definition-by-Ltac involving the `refine` tactic, as mentioned *supra* in the present section.

Quotient of an Equivalence Relation

Another subtlety involved in type theory is the treatment of equivalence relations. Usual mathematical reasoning often uses the notion of quotient set to define various objects in abstract algebra, including the construction of the field of reals. Yet this notion of quotient does not fit easily into type theory. Instead, we often consider a pair (T, eq) composed with a type T and an equivalence relation on T, the whole being called a setoid. (Likewise, the constructive axiomatization of the reals performed in C-CoRN relies on constructive setoids.) One such formalism thus requires to explicitly deal with equivalence relations, and functions defined on setoids have to be shown compatible with respect to the corresponding equivalence relation.

In the Coq system, a systematic machinery has been developed to handle these notions, especially the Setoid-rewriting-tactic. For instance, the propositional equivalence `iff` (denoted by '`<->`') can obviously be viewed as an equivalence relation, and in recent versions of Coq it is possible to rewrite propositions with lemmas using this connector with the help of the `rewrite` tactic, just as well we can rewrite Boolean propositions with lemmas using the bare Leibniz equality.

Last but not least, we can mention the "Quotient Interface" developed in [36] that makes it possible to obtain high-level interfaces for *constructive quotients on types with decidable equality*, which falls in very well with the kind of structures provided by the SSReflect libraries.

3.2.6 Around the Certificate-Based Approach

In this section we will summarize the different approaches that are possible when working on the formal verification of algorithms. We will especially focus on the notion of "certificate".

The Certificate-Based Approach. First, to undertake the verification of costly algorithms, a natural approach relies on the concept of *certificates* (also known as *witnesses*). They are typically useful when it is more suitable to check the outputs produced by a given algorithm, than to formally prove the algorithm itself. This may be the case when the considered algorithm relies on a large amount of computation, possibly with backtracking, heuristic strategies and approximations.

With this approach, we notably need to design a specific type of certificate, along with a *certificates checker*, which is typically a computable Boolean function defined within a Proof Assistant (PA). This requires the availability of a "*certifying property*" that characterizes the validity of the algorithm, e.g., in the case of the factorization of a big integer n into factors a and b, it will be sufficient to verify that the equality $n = a \times b$ holds inside the PA. Yet the main part of the computations corresponding to the algorithm at stake (here, factorization) will often be processed outside the PA, in a Computer Algebra System (CAS) or so.

Certifying Algorithms and Skeptical Approach. Depending on whether one takes the point of view of CAS or PA, this certificate-based approach coincides with the concept of *certifying algorithm* [116], or with the so-called *skeptical approach* (as opposed to the *believing* and *autarkic* approaches, these three terms having been coined in [7] and further developed in [6, 5]).

This skeptical methodology has been widely used in the past twenty years, involving the HOL Light proof assistant [76, 75, 74], as well as the Coq proof assistant: linking Coq and Maple [49, 51], developing primality certificates [68, 69, 163], as well as verifying SAT/SMT-based certificates [2].

The main advantage of the skeptical approach is that the highly computational part of the certifying algorithm can be executed on a highly efficient computation platform, with as many error-prone optimizations as desired.

Still, this approach has to be validated: we need to design "good" certificates that are of reasonable size and can be easily checked with the formalized checker. Then this checker should have been formally proved, using an appropriate "*certifying property*" that characterizes the problem at stake. Note finally that the *bottleneck* of this approach is that its correctness relies on the individual verification of the output for all considered instances of the overall problem, whose number will have to be estimated.

Autarkic Approach. Contrastingly, another methodology called *autarkic approach* relies on a formalization where all computations are carried out in the formal proof assistant (PA) itself. This approach is clearly more computationally expensive than the skeptical approach, so it may be more suitable when focusing on algorithms that do not involve much backtracking and can be efficiently implemented in the PA.

Extraction. Furthermore, when a given algorithm has been implemented in a PA such as Coq, the availability of an extraction mechanism can be used to get a compilable program source from the formalized algorithm, which is correct by construction. This is for example the approach followed in the design on the CompCert C verified compiler [14, 106, 105]. Actually, such a mechanism of extraction can be used when relying on the autarkic approach as well as on the skeptical one, where the extracted program would just be a certificate checker.

Proof of Programs. Finally, another usual approach when dealing with the formal verification of an algorithm consists of formally proving that a given implementation of the algorithm does not raise any execution error and meets its specification, typically expressed in a specific language of annotations, based on *Hoare logic* [80]. In particular, there exist some state-of-the-art tools such as the platform Frama-C/Jessie/Why [56, 43] that allows one to formally verify C programs, by generating proof obligations that can be discharged by SMT tools and/or proof assistants.

While we will mostly focus on formally verifying the SLZ algorithm in this thesis (cf. Section 2.5), especially relying on the skeptical approach, the proof-of-programs approach might be more suitable for verifying the Lefèvre algorithm, given the huge number of *degree*-1 polynomial approximations that would be involved in the corresponding certificates.

3.2.7 Description of the Coq Libraries at Stake

The Coq Standard Library. Coq comes with a comprehensive set of libraries for Booleans, integers, rationals, real numbers, lists, etc. An online documentation is available at http://coq.inria.fr/stdlib/. In this section, we will mainly focus on some characteristics related to the library on real numbers:

The Reals Library. The design of the Reals library relies on the axiomatization of the properties of $\mathbb{R}$ as a complete Archimedean ordered field [115]. Moreover, it is a classical axiomatization, due to the presence of the axiom of total order (also known as *trichotomy*):

```
Axiom total_order_T :
  forall r1 r2 : R, {r1 < r2} + {r1 = r2} + {r1 > r2}.
```

This axiom, which can be seen as an instance of excluded middle in `Type`, can be used for instance to derive[7] the "decidability of arithmetical statements" `{forall n, P n} + {~forall n, P n}`, for any predicate "`P : nat -> Prop`" that is *decidable* (i.e., such that "`forall n, {P n} + {~P n}`" holds).

[7] See for example http://coq.inria.fr/stdlib/Coq.Reals.Rlogic.html

At first, the logic of Coq had sort `Set` impredicative (i.e., a term such as "`t : forall A : Set, A -> A`" still had type `Set`, allowing one to replace `A` with `t` itself, etc.), so that it was quite dangerous to deal with classical logic within Coq. Indeed, it was shown inconsistent to assume the excluded middle in `Set` (e.g., at the time `sumbool` was in `Set`) with `Set` impredicative [61]. And yet having a strong version of excluded middle in `Set` or `Type`, or at least the trichotomy axiom, can be really useful. For instance one can use this axiom, along with strong elimination in `Type`, to define some non-continuous functions from `R` to `R`, which are typical in numerical analysis [115].

As a matter of fact, the sort `Set` is now predicative since Coq-v8, given that its underlying logic (often called pCIC for predicative Calculus of Inductive Constructions) does not have the typing rule for `Set` impredicative any more. In some sense, the CIC has been weakened in such a way that it allows one to safely cope with classical logic and any other similar axiomatics.

C-CoRN. Another kind of formalization of real numbers is provided in the Constructive Coq Repository at Nijmegen (C-CoRN), which is a library of constructive mathematics that originated in the FTA project for formalizing the Fundamental Theorem of Algebra. We recall that this theorem says that any non-constant polynomial on the field of complex numbers has at least one root. The C-CoRN library gathers an axiomatic algebraic hierarchy (including real and complex numbers) that is *intuitionistic*, and for which Milad Niqui constructed a concrete model to demonstrate the soundness of all axioms involved in C-CoRN's real number structure [135].

Note that in an intuitionistic setting, equality on real numbers is semi-decidable: for any particular pair of constructive real numbers (x, y), it is always possible to prove that $x \neq y$ if it is the case, while in general we cannot decide that $x = y$ in finite time. Intuitively, if we consider the reals x and y as infinite Cauchy sequences of rationals (x_n), (y_n), we should prove that we have $\lim_{n\to+\infty}(x_n - y_n) = 0$, or equivalently that for any $n \in \mathbb{N}^*$, there exists a threshold $N_n \in \mathbb{N}$ such that for all $k \geqslant N_n$, we have $|x_k - y_k| < 1/n$, which cannot be decided within a finite amount of computation.

In particular, C-CoRN's formalism deals with constructive setoids $(A, \#, \equiv)$ where $\equiv$ is an equivalence relation and $\#$ is a relation called apartness. Classically speaking, apartness coincides with the *negation of equality*, but in an intuitionistic point of view, it is a more primitive concept, as suggested by the remark above on Cauchy sequences, and equality is defined as the *negation of apartness*.

Furthermore, O'Connor [137] developed some computable exact reals in Coq on top of the C-CoRN library, called `fastreal`. Then Krebbers [98] developed a new library of computable exact reals in Coq using machine integers to increase the efficiency of computations.

The SSReflect Extension. The SSReflect extension of Coq refers to a formal proof methodology called *small scale reflection*, which is fully described in [65]. We can relate this methodology to the concept of reflection, which consists of reducing a whole logical proof to a single proof step composed with a *computation* on a symbolic representation of the problem and the invocation of a correctness lemma, along with the required transformations between the

logical and symbolic representations. This concept is implemented, for example, in the `field` tactic for solving equations over a given field structure [50], and in `romega`, the reflexive variant of the `omega` tactic for Presburger arithmetic. In SSReflect we actually encounter this approach everywhere, especially in the low-level contents of the formalized theories. For example, every decidable predicate is directly formalized using a Boolean function "`p : T -> bool`," rather than using a logical predicate "`p' : T -> Prop`" along with a decidability lemma "`p'_dec : forall x : T, {p' x} + {~ p' x}`." Here, the symbolic representation is just based on the inductive type

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

while in practice one can consider more sophisticated inductive types.

Moreover, SSReflect provides a small set of quite powerful tactics (`move`, `case`, `elim`, `apply`, `rewrite`, etc.) which are highly customizable with the help of tacticals (including '`:`', '`=>`', '`/=`', '`//`', '`//=`'). This often leads to more "compact" proof lines that tend to shorten the proof scripts, and also to make the structure of proofs more visible. For example, the SSReflect proof sentence "`elim: s=> [//|x s' IH] /=`" could roughly be written in plain Coq this way: "`elim s; clear s; [trivial|intros x s' IH]; simpl`".

SSReflect also provides a full support for forward chaining thanks to the tactics `have`, `suff`, and `wlog`.

While SSReflect originated in the formalization of the Four-Colour Theorem carried out by G. Gonthier and B. Werner [63], this extension is maintained by the project-team Math-Components at the Microsoft Research–INRIA Joint Centre[8] and it comes with a set of Coq libraries developed upon the SSReflect extension, focusing on various aspects of abstract algebra such as polynomials, matrices and finite groups.

The SSReflect libraries make heavy use of *canonical structures* [150] as well as implicit arguments. Thus a SSReflect library typically begins by the invocation

```
Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

and these libraries follow systematic naming conventions as well as the SSReflect proving guidelines (including the use of *bullets* and *terminators*), as described in [64].

Coq Libraries on Floating-Point Arithmetic. The publication of the IEEE-754 standard for floating-point (FP) arithmetic provided a precise description of the FP formats and operations, which paved the way for a predictable, fully-specified, FP arithmetic, and is amenable to formal methods. In the sequel, we will focus on the formalizations of FP arithmetic that have been developed within the Coq proof assistant. The formalization [109] focused on a formal description of the standard following a low-level approach, which made it possible to formally verify a FP adder. Then, some high-level formalizations

[8] http://www.msr-inria.inria.fr/Projects/math-components/

of FP arithmetic have been designed in Coq, namely the Float library [45], the Pff library [21, 20, 19], and the Flocq library [22].

Pff (in French, "Preuves Formelles sur les Flottants") was built upon the Float library, and a significant number of theorems have been added subsequently. The formalism of Pff relies on FP formats with gradual underflow. To be more precise, a FP number $(m, e) \in \mathbb{Z} \times \mathbb{Z}$ (representing the value $m \cdot \beta^e$) is said to belong to the FP format parameterized by $(\beta^p, E) \in \mathbb{N}^* \times \mathbb{N}$ if the following condition holds:

$$|m| < \beta^p \quad \wedge \quad e \geqslant -E.$$

Flocq (floats for Coq) provides a library for multi-radix, multiple-precision FP arithmetic in Coq. It provides generic definition of valid formats and roundings modes, as well as specialized definitions for the usual ones. It includes the fixed-point (FIX), floating-point with unbounded exponents (FLX), floating-point with gradual underflow (FLT), and floating-point with flush-to-zero (FTZ) formats. Note that the FLT format is equivalent to the only kind of format available in Pff, so that Flocq encompasses the formalism of Pff. Moreover, Flocq provides not only axiomatic definitions of rounding that are suitable for proving high-level properties, but also a computable version of the common roundings modes (DN, UP, NE, and so on). Based on this machinery, Flocq offers effective arithmetic operators for addition, multiplication, division and square root. Some of these features of Flocq were directly inspired by the CoqInterval library for interval arithmetic, whose core is now based on Flocq.

The CoqInterval library is designed in a modular way; it defines intervals with floating-point bounds in an abstract way and provides a concrete implementation for some elementary functions such as exp and arctan, based on Taylor series, interval arithmetic, and if desired, machine integers. This allows one to quickly compute an interval enclosure "$f(\boldsymbol{I})$" for these functions.

Likewise, the libraries [137] and [98] provide a multiple-precision real arithmetic but the main difference between them and CoqInterval/Flocq is that in the latter libraries, the precision is handled explicitly in all FP routines, while in the former libraries, the user just specifies the desired output precision for the final result and the exact reals involved in intermediate calculations are internally approximated with a sufficient precision.

Finally, the formal certification method chosen in all the above libraries consists of linking the "computable reals" to an axiomatic formalization of real numbers (either Reals for the Float, Pff, Flocq and CoqInterval libraries, or C-CoRN). Thus doing a formal proof of correctness will amount to saying the implementation is correct with respect to a given abstract, mathematical object defined in the system.

Part II

Contributions

Chapter 4

Rigorous Polynomial Approximation in the Coq Formal Proof Assistant

The work presented in this chapter, whose first results were published in the proceedings of the NASA Formal Methods 2012 conference [28], is the fruit of a collaboration between members of the TaMaDi project. The formal developments presented here were carried out mainly by Ioana Paşca, Micaela Mayero and myself, with remarks and suggestions by Laurent Théry and Laurence Rideau.

4.1 Rigorous Approximation of Functions by Polynomials

It is frequently useful to be able to replace a given function of a real variable by a simpler function, such as a polynomial, chosen to have values very close to those of the given function, since such an approximation may be more compact to represent and store but also more efficient to evaluate and manipulate. As long as evaluation is concerned, polynomial approximations are especially important. In general the basic functions that are implemented in hardware on a processor are limited to addition, subtraction, multiplication, and sometimes division. Moreover, division is significantly slower than multiplication. The only functions of one variable that one may evaluate using a bounded number of additions/subtractions, multiplications and comparisons are piecewise polynomials: hence, on such systems, polynomial approximations are not only a good choice for implementing more complex mathematical functions, they are frequently the only one that makes sense.

Polynomial approximations for widely used functions used to be tabulated in handbooks [1]. Nowadays, most computer algebra systems provide routines for obtaining polynomial approximations of commonly used functions. However, when bounds for the approximation errors are available, they are not guaranteed to be accurate and are sometimes unreliable.

Our goal is to offer formally verified error bounds for such polynomial approximations. To this end, we formalize in Coq some symbolic-numeric techniques

related to the concept of *rigorous polynomial approximations* [88], that is, polynomial approximations for which (i) a provably-not-underestimated error bound is provided, (ii) the framework is suitable for formal proof. We especially focus on genericity (for our formalization to be applicable to a large class of problems) and we follow the *autarkic* approach: all the computations are performed in the formal proof assistant. We will thus devote a particular care to the efficiency of computations.

4.1.1 Motivations

Most numerical systems depend on standard functions like exp, sin, etc., which are implemented in libraries called `libm`s, e.g.: `CRlibm`, `glibc`, Sun `libmcr` or Intel's `libm`. These `libm`s must offer guarantees regarding the provided accuracy: they are of course heavily tested before being published, but for precisions higher that single precision, an exhaustive test is impossible [52]. Hence a proof of the behavior of the program that implements a standard function should come with it, whenever possible. One of the key elements of such a proof would be the guarantee that the used polynomial approximation is within some threshold from the function. This requirement is even more important when *correct rounding* is at stake. Most `libm`s do not provide correctly-rounded functions, although the IEEE 754-2008 Standard for Floating-Point (FP) Arithmetic [85] recommends it for a set of basic functions. Implementing a correctly-rounded function requires rigorous polynomial approximations at two important steps: when actually implementing the function in a given precision, and—before that—when trying to solve the *Table Maker's Dilemma* for that precision (cf. Section 2.4).

In particular, a central goal of the TaMaDi project [126] aims at safely computing the hardest-to-round points for the most common functions and formats. Doing this requires very accurate polynomial approximations that are formally verified.

Beside the Table Maker's Dilemma, the implementation of correctly rounded elementary functions is a complex process, which includes finding polynomial approximations for the considered function that are accurate enough to allow for correct rounding. Obtaining good polynomial approximations is detailed in [27, 26, 31]. In the same time, the approximation error between the function and the polynomial is very important since one must make sure that the approximation is good enough. The description of a fast, automatic and verifiable process was given in [88].

In the context of implementing a standard function, we are interested in finding polynomial approximations for which, given a degree n, the maximum error between the function and the polynomial is minimum: this "minimax approximation" has been broadly developed in the literature and its application to function implementation is discussed in detail in [31, 127]. Usually this approximation is computed numerically [146], so an *a posteriori* error bound is needed. Obtaining a tight bound for the approximation error reduces to computing a tight bound for the supremum norm of the error function over the considered interval. Absolute error as well as relative errors can be considered. For the sake of simplicity, in this work, we consider absolute errors only (relative errors would be handled similarly). Our problem can be seen as a univariate rigorous global optimization problem, however, obtaining a tight and formally verified interval bound for the supremum norm of the error function presents issues unsuspected

at a first sight [33], so that techniques like interval arithmetic and Taylor models are needed. An introduction to these concepts is given below.

Interval arithmetic and Taylor models. The usual arithmetic operations and functions are straightforwardly extended to handle intervals. One use of interval arithmetic is bounding the image of a function over an interval. Interval calculations frequently overestimate the image of a function. This phenomenon is in general proportional to the width of the input interval. We are therefore interested in using thin input intervals in order to get a tight bound on the image of the function. While subdivision methods are successfully used in general, when trying to solve this problem, one is faced with what is known in the literature of interval-based methods as a "dependency phenomenon": since function f and its approximating polynomial P are highly correlated, branch and bound methods based on using intervals of smaller width to obtain less overestimation, end up with an unreasonably high number of small intervals. To reduce the dependency, *Taylor models* are used. They are a basic tool for replacing functions with a polynomial and an interval remainder bound, on which basic arithmetic operations or bounding methods are easier.

4.1.2 Related Work

Taylor models [110, 134, 111] are used in rigorous global optimization problems [110, 12, 33, 13] and validated solutions of ODEs [132] with applications to critical systems like particle accelerators [13] or robust space mission design [108]. Freely available implementations are scarce. One such implementation is available in Sollya [32], which handles univariate functions only, but provides multiple-precision support for the coefficients. It was used for proving the correctness of supremum norms of approximation errors in [33]. However, this remains a C implementation that does not provide formally proved Taylor models, although this would be necessary for having a completely formally verified algorithm.

There were several attempts to formalize Taylor models in proof assistants. An implementation of multivariate Taylor models is presented in [166]. They are implemented on top of a library of exact real arithmetic, which is more costly than FP arithmetic. Also, the purpose of that work is different than ours. It is appropriate for multivariate polynomials with small degrees, while we want univariate polynomials and high degrees. There are no formal proofs for that implementation. The work in [30] presents an implementation of univariate Taylor models in the PVS theorem prover. Though formally proved, this implementation contains ad-hoc models for only a few functions (exp, sin, arctan) and it is not efficient enough for our needs, as it is unable to produce Taylor models of degree higher than 6. The work in [37] presents another formalization of Taylor models in Coq. It uses polynomials with FP coefficients. However, the coefficients are axiomatized, so we cannot compute the actual Taylor model in that implementation. We can only talk about the properties of the involved algorithms.

4.1.3 Outline

Our goal is to provide a modular implementation of univariate Taylor models in Coq, which is efficient enough to produce very accurate approximations of elementary real functions. We start by presenting in Section 4.2 the mathematical definitions of Taylor models as well as efficient algorithms used in their implementation. We then present in Section 4.3 the Coq implementation itself, whose performances (in terms of efficiency as well as accuracy) are briefly evaluated in Section 4.4. Finally we summarize in Section 4.5 the main issues we have encountered during the formal verification of our implementation, before drawing some perspectives in Section 4.6.

4.2 Presentation of the Notion of Taylor Models

4.2.1 Definition, Arithmetic

A Rigorous Polynomial Approximation (RPA) of order n for a function f that is supposed to be $n+1$ times differentiable over an interval $[a,b]$, is a pair $(T, \mathbf{\Delta})$ formed by a polynomial T of degree n, and an interval part $\mathbf{\Delta}$, such that $\forall x \in [a,b],\ f(x) - T(x) \in \mathbf{\Delta}$. The polynomial can possibly be a Taylor expansion of the function at a given point, in which case the RPA will be called a Taylor Model (TM). And the interval $\mathbf{\Delta}$ (called an enclosure of the approximation error between the polynomial T and the function that it approximates.

For usual functions, we can easily build a TM by putting together a Taylor expansion of the function with an error bound deduced from the Taylor–Lagrange formula, relying on the recurrence relations satisfied by successive derivatives of the functions (see Section 4.2.3). When using the same approach for composite functions, the error we get for the remainder is too pessimistic [33]. Hence the usefulness of considering operations on the TMs themselves: simple algebraic rules like addition, multiplication and composition are applied recursively on the structure of function f, and build corresponding TMs step by step to finally provide a TM for f over $[a,b]$. Usually, the use of these operations on TMs offers a much tighter error than the one directly computed for the whole function [33].

For example, the addition of TMs is defined as follows: let $(P_1, \mathbf{\Delta}_1)$ and $(P_2, \mathbf{\Delta}_2)$ be two TMs of order n for two functions f_1 and f_2, over $[a,b]$. The sum of the two models is an order-n TM for $f_1 + f_2$ over $[a,b]$ and is obtained by adding the two polynomials and the remainder bounds: $(P_1, \mathbf{\Delta}_1) + (P_2, \mathbf{\Delta}_2) = (P_1 + P_2, \mathbf{\Delta}_1 + \mathbf{\Delta}_2)$. For multiplication and composition, similar rules are defined.

We follow the definitions from [88, 33], and represent the polynomial T with *tight interval coefficients*. This choice is motivated by the ease of programming (rounding errors are directly handled by the interval arithmetic) and also by the fact that we want to ensure that the true coefficients of the Taylor polynomial lie inside the corresponding intervals. This is essential for applications that need to handle removable discontinuities [33]. For our formalization purpose, we recall and explain briefly in what follows the definition of valid Taylor models [88, Def. 2.1.3], and refer to [88, Chap. 2] for detailed algorithms regarding operations with Taylor models for univariate functions.

4.2.2 Valid Taylor Models

A Taylor Model (TM) for a function f is a pair $(T, \boldsymbol{\Delta})$. The relation between f and $(T, \boldsymbol{\Delta})$ can be rigorously formalized as follows.

Definition 4.1. Let $f : \boldsymbol{I} \to \mathbb{R}$ be a function, $\boldsymbol{x_0}$ be a small interval around an expansion point x_0. Let T be a polynomial with interval coefficients $\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}$ and $\boldsymbol{\Delta}$ an interval. We say that $(T, \boldsymbol{\Delta})$ is a Taylor model of f at $\boldsymbol{x_0}$ on $\boldsymbol{I}$ when

$$\begin{cases} \boldsymbol{x_0} \subset \boldsymbol{I}, \\ 0 \in \boldsymbol{\Delta}, \\ \forall \xi_0 \in \boldsymbol{x_0}, \exists \alpha_0 \in \boldsymbol{a_0}, \ldots, \alpha_n \in \boldsymbol{a_n}, \forall x \in \boldsymbol{I}, \exists \delta \in \boldsymbol{\Delta}, f(x) - \sum_{i=0}^{n} \alpha_i \left(x - \xi_0\right)^i = \delta. \end{cases}$$

Informally, this definition says that there is always a way to pick some values α_i in the intervals $\boldsymbol{a_i}$ in such a way that the difference between the resulting polynomial and f around $\boldsymbol{x_0}$ is contained in $\boldsymbol{\Delta}$. This validity is the invariant that is preserved when performing operations on Taylor models (addition, multiplication, etc.). Obviously, once a Taylor model $(T, \boldsymbol{\Delta})$ is computed, if need be, one can get rid of the interval coefficients $\boldsymbol{a_i}$ in T by picking arbitrary α_i and accumulating in $\boldsymbol{\Delta}$ the resulting errors.

4.2.3 Computing the Coefficients and the Remainder

We are now interested in an automatic way of providing both the coefficients $\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}$ and $\boldsymbol{\Delta}$ of Definition 4.1 for basic functions. It is classical to use the following

Theorem 4.1 (Taylor–Lagrange Formula). *If f is $n+1$ times differentiable on a domain $\boldsymbol{I}$, then we can expand f in its Taylor series around any point $x_0 \in \boldsymbol{I}$ and we have: $\forall x \in \boldsymbol{I}$, $\exists \xi$ between x_0 and x such that*

$$f(x) = \underbrace{\left(\sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!} \left(x - x_0\right)^i \right)}_{T(x)} + \underbrace{\frac{f^{(n+1)}(\xi)}{(n+1)!} \left(x - x_0\right)^{n+1}}_{\Delta(x,\xi)}.$$

Computing interval enclosures $\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}$, for the coefficients of T, reduces to finding enclosures of the first n derivatives of f at x_0 in an efficient way. The same applies for computing $\boldsymbol{\Delta}$, based on an interval enclosure of the $(n+1)$th derivative of f over $\boldsymbol{I}$. However, the expressions for successive derivatives of practical functions typically become very involved with increasing n. Fortunately, it is not necessary to generate these expressions for obtaining values of $\{f^{(i)}(x_0), i = 0, \ldots, n\}$. For basic functions, formulas are available since Moore [125] (see also [70]). There one finds either recurrence relations between successive derivatives of f, or a simple closed formula for them. And yet, this is a case-by-case approach, and we would like to use a more generic process, which allows us to deal with a broader class of functions in a more uniform way suitable to formalization.

Recurrence Relations for D-finite Functions. An algorithmic approach exists for finding recurrence relations between the Taylor coefficients for a large class of functions that are solutions of linear ordinary differential equations (LODE) with polynomial coefficients, usually called *D-finite functions*. The Taylor coefficients of these functions satisfy a linear recurrence with polynomial coefficients [156]. Most common functions are *D-finite*—it is estimated that around 60% of the functions described by Abramowitz & Stegun in [1] belong to this category—, while a simple counter-example is tan, whose ODE is not linear. For any D-finite function it is possible to generate the recurrence relation directly from the differential equation that defines the function, see for example the Gfun module in Maple [151]. From the recurrence relation, the computation of the first n coefficients is done in linear time.

Let let us take a simple example and consider the function

$$f = \exp$$

It satisfies the LODE

$$f' = f \qquad f(0) = 1$$

hence the following recurrence on the Taylor coefficients of f around the origin:

$$(n+1)c_{n+1} = c_n \qquad c_0 = 1,$$

whose solution can be defined by: $\forall n \in \mathbb{N},\ c_n = \frac{1}{n!}$.

This property lets us include in the class of *basic functions* all the D-finite functions. We will see in Section 4.3.1 that this allows us to provide a uniform and efficient approach for computing Taylor coefficients, suitable for formalization. We note that our data structure for that is *recurrence relation + initial conditions* and that the formalization of the isomorphic transformation from the *LODE + initial conditions*, used as input in Gfun is subject of future research.

4.3 Implementation of Taylor Models in Coq

We provide an implementation[1] of Taylor models that is efficient enough to produce very accurate approximation polynomials in a reasonable amount of time. Moreover, the work is carried out in the Coq proof assistant, which provides a formal setting where the implementation can then be formally verified (see Section 4.5).

One of our goals with this implementation is to be as generic as possible. As pointed out in the beginning of Section 4.2.1, we notice that a Taylor model (TM) is just an instance of the more general notion of *rigorous polynomial approximation* (RPA). For a function f, a RPA is a pair $(T, \mathbf{\Delta})$ where T is a polynomial and $\mathbf{\Delta}$ an interval containing the approximation error between f and T. We can choose a Taylor polynomial for T and thus obtain a TM, but other kinds of approximation are also available, including Chebyshev models which are based on Chebyshev polynomials. This generic rigorous polynomial approximation structure will look like:

[1] The Coq development is available at http://tamadi.gforge.inria.fr/CoqApprox/

```
Structure rpa := { approx: polynomial; error: interval }
```

In this structure, we also want genericity not only for `polynomial` with respect to the type of its coefficients and to its physical implementation but also for the type for intervals. Users can then experiment with different combinations of datatypes. Also, this genericity allows us to factorize our implementation and contributes to facilitate the proofs of correctness.

We implement Taylor models as an instance of a generic RPA following what is presented in Section 4.2. Our development relies on the following tools and libraries:

- the SSReflect extension of Coq,
- the CoqInterval library, which is based on
- the Flocq library, itself based on
- the Reals library from the Coq standard library.

All these libraries have been presented in Section 3.2.7. In particular, we recall that the formal verification method chosen in both Flocq and CoqInterval consists of linking the effective algorithms (in charge of producing FP approximations or interval enclosures) to a mathematical formalization of real numbers (the Reals library). So doing a formal proof of correctness amounts to saying an implementation is correct with respect to a given abstract, mathematical object defined in the system. We want to follow the same idea in our Taylor model development: implement a computable Taylor model for a given function and formally prove its correctness with respect to the abstract formalization of that function in Coq. This can be done by using Definition 4.1 and the axiomatized real-valued functions from the Coq standard library.

We recall there are some restrictions to the programs that can be executed in Coq: they must always terminate and must be purely functional, i.e., no side-effects are allowed. Nevertheless, it is possible to define some algorithms in Coq on top of the multiple-precision arithmetic libraries `BigN` or `BigZ`, based on the binary trees described in [68]. This allows one to benefit from the machine modular arithmetic (32- or 64-bits depending on the machine) for computing in Coq.

Thus following the *autarkic* approach for our development of Taylor models, we have to consider polynomials with coefficients that are a suitable kind of computational objects. As described in Section 4.2, we use *intervals with FP bounds,* by relying on the CoqInterval library [118] that provides such datatypes and related algorithms, which can possibly be instantiated with machine integers (`BigZ`). By choosing a functional implementation for polynomials (e.g., lists), we then obtain Taylor models that are directly executable within Coq. In the next section we describe in detail this modular implementation.

4.3.1 A Modular Implementation of Taylor Models

The Coq proof assistant comes with three different mechanisms for modularization: *type classes* [155], *canonical structures* [150], and *modules* [35, 34]. Modules are less generic than the other two (which are first-class citizens) but they have a better computational behavior. Indeed, module applications are performed

statically, so the code that is executed is often more compact. Since the objects to be formalized only require simple parametricity, we have chosen to use modules for developing our generic implementation. First, abstract interfaces called `Module Type`s are defined. Then concrete "instances" of these abstract interfaces are created by providing an implementation for all the fields of the `Module Type`. Furthermore, the definition of `Module`s can be parameterized by other `Module`s. These parameterized modules are crucial to factorize code in our data structures.

Abstract Polynomials, Coefficients and Intervals.

We describe abstract interfaces for *polynomials* and for their *coefficients* using Coq's `Module Type`. The interface for coefficients contains the common base of all the computable real numbers we may want to use. Usually coefficients of a polynomial are taken in a ring. We cannot do this here. For example, the addition of two intervals is not associative. Therefore, the abstract interface for coefficients contains only the required operations (addition, multiplication, etc.) where some basic properties (associativity, distributivity, etc.) are ruled out. The case of abstract polynomials is similar. They are also a `Module Type` but this time parameterized by the coefficients. The interface contains only the operations on polynomials (addition, evaluation, iterator, etc.) with the properties that are satisfied by all common instantiations of polynomials.

For intervals, we directly use the abstract interface provided by the CoqInterval library [118].

Rigorous Polynomial Approximations.

We are now able to give the definition of our rigorous polynomial approximation.

```
Module RigPolyApprox (C : BaseOps)(P : PolyOps C)(I : IntervalOps).
Structure rpa : Type := RPA { approx : P.T;  error : I.type }.
```

The module is parameterized by `C` (the coefficients), by `P` (the polynomials with coefficients in `C`), and by `I` (the intervals).

Generic Taylor Polynomials.

Before implementing our Taylor models, we use the abstract coefficients and polynomials to implement generic Taylor polynomials. These polynomials are computed using an algorithm based on recurrence relations as described in Section 4.2.3. This algorithm can be implemented in a generic way. It takes as argument the relation between successive coefficients, the initial conditions and outputs the Taylor polynomial.

We detail the example of the exponential, which was also presented in Section 4.2.3. The Taylor coefficients $(c_n)_{n\in\mathbb{N}}$ satisfy the recurrence

$$(n+1)c_{n+1} - c_n = 0 \quad \Longleftrightarrow \quad c_{n+1} = \frac{c_n}{n+1}.$$

The corresponding Coq code is

```
Definition exp_rec (u : T) (n : nat) : T := tdiv u (tnat n).
```

where `tdiv` is the division on our coefficients and `tnat` is an injection of integers to our type of coefficients. We then implement the generic Taylor polynomial for the exponential around a point x_0 with the following definition:

```
Definition T_exp (x0 : C.T) (n : nat) := trec1 exp_rec (texp x0) n.
```

In this definition, `trec1` is the function in the `PolyOps` interface that is in charge of producing a polynomial of degree `n` from a recurrence relation of order 1 (here, `exp_rec`) and an initial condition (here, "`texp x0`," the value of the exponential at `x0`). The interface also contains `trec2` and `trecN` for producing polynomials from recurrences of order 2 and order N, with the appropriate number of initial conditions, in an efficient way: they are implemented by tail-recursive functions, and having specific functions for recurrences of order 1 and 2 makes it possible to have optimized implementations for these frequent recurrences. As a matter of fact, all the functions we currently dispose of in our library are defined using `trec1` and `trec2`. We provide generic Taylor polynomials for constant functions, identity, $x \mapsto \frac{1}{x}$, $\sqrt{\cdot}$, $\frac{1}{\sqrt{\cdot}}$, exp, ln, sin, cos, arcsin, arccos, arctan.

Taylor Models.

We implement Taylor models on top of the RPA structure by using polynomials with coefficients that are intervals with FP bounds, according to Section 4.2. Yet we are still generic with respect to the effective implementation of polynomials. For the remainder, we also use *intervals with FP bounds*. Note that this datatype is provided by the CoqInterval library [118], whose design is also based on modules, in such a way that it is possible to plug all the machinery on the desired kind of Coq integers (i.e., `Z` or `BigZ`).

In a Taylor model for a basic function (exp, sin, etc.), polynomials are instances of the generic Taylor polynomials implemented with the help of recurrence relations described above. The remainder is computed with the help of the Taylor–Lagrange formula in Lemma 4.1. For this computation, thanks to the parameterized module, we reuse the generic recurrence relations. The order-n Taylor model for the exponential on interval `X` expanded at the small interval `X0` is as follows:

```
Definition TM_exp (X0 X : T) (n : nat) :=
  RPA (T_exp X0 n) (Trem T_exp X0 X n).
```

We implement Taylor models for the addition, multiplication, and composition of two functions by arithmetic manipulations on the Taylor models of the two functions, as described in Section 4.2. Here is the example of addition:

```
Definition TM_add (Mf Mg : rpa) : rpa :=
  RPA (Pol.tadd (approx Mf) (approx Mg))
      (I.add (error Mf) (error Mg)).
```

The polynomial approximation is just the sum of the two approximations and the interval error is the sum of the two errors. Multiplication is almost as intuitive. We consider the truncated multiplication of the two polynomials and we make sure that the error interval takes into account the remaining parts of the truncated multiplication. Composition is more complex. It uses addition and multiplication of Taylor polynomials. Division of Taylor models is implemented in term of multiplication and composition with the inverse function $x \mapsto 1/x$. The corresponding algorithms are fully described in [88].

Figure 4.1 summarizes the hierarchy of the computational part[2] of the library. This involves four abstract interfaces: `BaseOps` for the three base operations $+$, $-$, $\times$, `PolyOps` for polynomials, which is parameterized by the implementation of coefficients, `IntervalOps`, which is provided by CoqInterval, and `FullOps`, which extends `BaseOps` with other operations such as division, exponentiation, and elementary functions (e.g., the exponential). It can be noted that this latter interface `FullOps` will be instantiated either with abstract functions based on the `Reals` library, or with computational functions, such as the evaluator `I.exp` for the exponential that is available in CoqInterval.

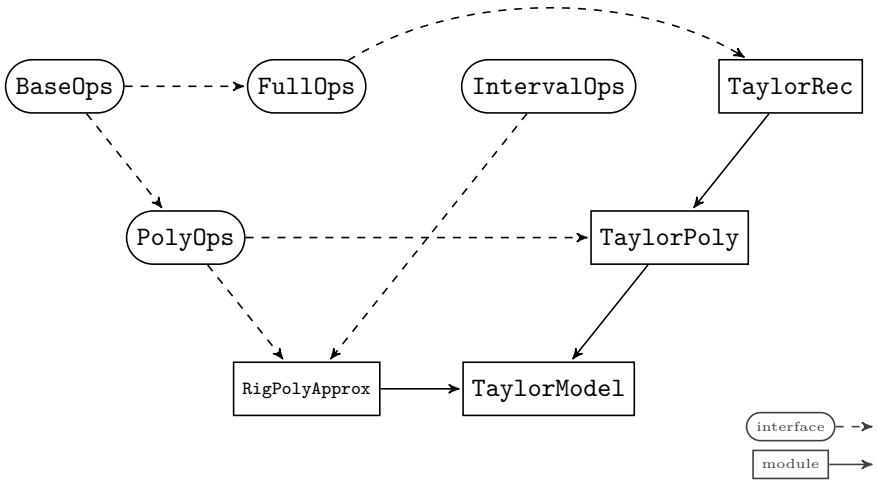

Figure 4.1 – *Overview of the modular hierarchy implemented in CoqApprox*

4.4 Some Preliminary Benchmarks

Given that we follow the so-called autarkic approach (see Section 3.2.6), we wanted to evaluate the performances of our Coq implementation of Taylor models, before starting proving anything. For this we compare our implementation to that of Sollya [32], a tool specially designed to handle such numerical approximation problems.

The Coq Taylor models we use for our tests are implemented with polynomials represented as simple lists with a linear access to its coefficients. The coefficients of the approximation polynomial in our instantiation of Coq Taylor models as well as the interval errors are implemented by intervals with multiple-precision FP bounds, as available in the CoqInterval library described in [118]. Note that because we need to compute the initial conditions for recurrences, only the basic functions already implemented in CoqInterval can have their corresponding Taylor models.

In Sollya, polynomials have interval coefficients and are represented by a (coefficient) array of intervals with multiple-precision FP bounds.

[2] Other modules have been added for the elaboration of the correctness proofs.

Timings, Accuracy and Comparisons

We compare the Coq and the Sollya implementations presented above on a selection of several benchmarks. Table 4.1 gives the timings as well as the tightness obtained for the remainders, taking the rounding errors into account in addition to the method error.

These benchmarks have been computed on a 8-core computer, Intel(R) Xeon(R) CPU E5520 @ 2.27GHz with 16 GB of memory, using Coq 8.3pl4 (with the `vm_compute` evaluation strategy), and Sollya 3.0.

	Execution time			Approximation error	
	Coq	Sollya	Ratio	Coq	Sollya
exp prec=1000, deg=70 $\boldsymbol{I}$=[127/128, 1]	0.716s	0.093s	7.7	1.80×2^{-906}	1.79×2^{-906}
sin prec=1000, deg=70 $\boldsymbol{I}$=[127/128, 1]	2.636s	0.088s	30	1.45×2^{-908}	1.44×2^{-908}
arctan prec=1000, deg=118 $\boldsymbol{I}$=[127/128, 1]	2.969s	0.420s	7.1	1.71×2^{-913}	1.30×2^{-967}
exp × sin prec=400, deg=20 $\boldsymbol{I}$=[127/128, 1]	0.812s	0.013s	63	1.36×2^{-222}	1.36×2^{-222}
exp × sin prec=400, deg=40 $\boldsymbol{I}$=[127/128, 1]	1.736s	0.040s	44	1.01×2^{-397}	1.53×2^{-397}
exp ∘ sin prec=400, deg=20 $\boldsymbol{I}$=[127/128, 1]	7.165s	0.011s	650	1.56×2^{-192}	1.83×2^{-192}
exp ∘ sin prec=400, deg=40 $\boldsymbol{I}$=[127/128, 1]	52.687s	0.065s	810	1.88×2^{-385}	1.38×2^{-384}

Table 4.1 – *Benchmarks and timings for our implementation in Coq*

Each cell of the first column of Table 4.1 contains a target function, the working precision in bits used for the computations, the order of the TM, and the interval under consideration. Each TM is expanded at the middle of the interval. For greater convenience, the errors in the last two columns are given using three decimal digits times a power of 2.

We notice that on these examples of TMs for *base functions* exp, sin and arctan, Coq is 7 to 30 times slower than Sollya, which is reasonable. Moreover, the error bounds so obtained have a similar order of magnitude.[3] Using the newly designed evaluation strategy `native_compute` [15] instead of `vm_compute`, we have done some similar experiments that tend to indicate we can expect a 2x to 3x speedup, allowing one to compute TMs for base functions in Coq only 10 times slower than Sollya [28].

[3]Note however that in Sollya, the arctan is implemented with a more specific algorithm, which trades accuracy for efficiency with respect to D-finite recurrences.

The ratio gets larger when composition is used. One possible explanation is that composition implies lots of polynomial manipulations and the implementation of polynomials as simple lists in Coq may be too naive. An interesting alternative could be to use persistent arrays [3] to have more efficient polynomials. Another possible improvement is at algorithmic level: while faster algorithms for polynomial multiplication exist [60], currently in all TMs related works $O(n^2)$ naive multiplication is used, resulting in a composition in $O(n^3)$, which can roughly be noticed in the last two rows of Table 4.1. We could improve that by using a Karatsuba-based approach, for instance.

4.5 Formal Verification of Our Implementation of Taylor Models

We prove that our implementation of Taylor models is correct with respect to the formalization of real functions available in the Coq standard library Reals that we presented in Section 3.2.7.

The `TaylorModel Module` presented in Figure 4.1 also imports a version of Taylor polynomials defined with axiomatic real numbers as coefficients. These polynomials are meant to be used only in the formal verification, when linking the computable Taylor models to the corresponding functions on axiomatic real numbers. This link is given by Definition 4.1 of a valid Taylor model given in Section 4.2.2. The definition can be easily formalized in the form of a predicate `validTM`.

```
Definition validTM X0 X M (f : ExtendedR -> ExtendedR) :=
  I.subset X0 X /\
  contains (error M) 0 /\
  let N := tsize (approx M) in
  forall x0, contains X0 x0 -> exists P, tsize P = N /\
    ( forall k, (k < N) ->
      contains (tnth (approx M) k) (tnth P k) ) /\
    forall x, contains X x ->
      contains (error M) (f x - teval P (x - x0)).
```

where `ExtendedR` denotes the *option type* that is defined in the CoqInterval library by

```
Inductive ExtendedR : Set :=
  | Xnan : ExtendedR
  | Xreal : R -> ExtendedR.
```

The theorem of correctness for the Taylor model of the exponential `TM_exp` then establishes the link between the model and the exponential function `Xexp` (from CoqInterval), itself defined in terms of the exponential function from the `Reals` library:

```
Lemma TM_exp_correct :
  forall X0 X n,
  I.subset X0 X ->
  ( exists t, contains X0 t ) ->
  validTM X0 X (TM_exp X0 X n) Xexp.
```

Note that the correctness theorems for all the other base functions have the same shape. Thus, we wanted to focus on proofs that are generic, so that a new instantiation of the Taylor polynomials for a given function can be smoothly proved, relying on the generic proof.

We have managed to prove some generic theorems of this kind, applicable for TMs defined in terms of `trec1` and `trec2`. This formalization effort led us to prove Taylor's theorem for functions over $\mathbb{R}$. Then, we can derive, say, `TM_exp_correct` by applying the generic theorem `TM_rec1_correct`, and discharge the various hypotheses that are required, especially the fact that the function `f` at stake propagates the NaNs (i.e., "`f Xnan = Xnan`"), and that the recurrence "preserves" the successive derivatives of the considered function. The formalization process of this last kind of properties is almost completed, in particular we have formally verified TMs for exp, sin, cos, $x \mapsto \frac{1}{x}$, identity, and "constant" functions.

Finally, the algorithms for addition, multiplication and composition requires a separate proof. To this end, we closely follow the pen-and-paper proofs that are detailed in [88, Sect. 2.2.2], and this formalization effort is still work in progress (we have completed the proofs for addition and composition but some subgoals remain for the multiplication of TMs). Among the various issues that have arisen in this process, we can mention some peculiarities related to the NaNs, for instance: what should return the evaluation of a null polynomial at `Xnan`? besides, does the interval $[\xi_0, \xi_0]$ always contain ξ_0, notably when $\xi_0 =$ `Xnan`? Finally, a number of the proofs related to the derivation of functions in the formalism of Coq's standard library were long and fastidious. Some recent works in the domain may improve this aspect [103].

4.6 Conclusion and Perspectives

In this chapter, we have described an implementation of Taylor models in the Coq proof assistant. We have addressed the following issues: the first one is genericity. We wanted our implementation to be applicable to a large class of problems. This motivates our use of modules in order to get this flexibility. The second issue is efficiency. Working in a formal setting has some impact in terms of efficiency. Before starting to prove anything, it was then crucial to evaluate if the computational power provided by the Coq system was sufficient for our needs. The results given in Section 4.4 clearly indicate that our implementation is worth proving formally. The third issue was to design a framework with some high-level and generic proofs, in order to ease the validation of each new Taylor model added to the library. The process of formally verifying our implementation is still work in progress, but we believe it should be complete in a couple of weeks.

As we aim at a complete formalization, a more subtle issue concerns the Taylor models for the basic functions and in particular how the model and its corresponding function can be formally related. This can be done in an ad-hoc way, deriving the recurrence relation from the formal definition. An interesting future work would be to investigate a more generic approach, trying to mimic what is provided by the *Dynamic Dictionary of Mathematical Functions* [8] in a formal setting.

Moreover, we have pointed out that a necessary building block to be able to evaluate our formalized Taylor models is the availability of an interval-based

evaluator. Some functions are still missing in CoqInterval and are worth to be implemented, but at the same time we would like to investigate some techniques that may provide an alternative to the availability of such an evaluator [119], including *fixed-point theorems*.

Having Taylor models is a milestone in our overall goal of getting formally proved hard-to-round cases for common functions and formats. Much more work needs to be done. A natural next step is to couple our models with some positivity test for polynomials, for example some *sums-of-squares* technique. This would give an automatic way of verifying polynomial approximations formally. It would also provide another way of evaluating the quality of our Taylor approximations. If they reveal to be not accurate enough for our needs, we could always switch to some better kinds of approximations such as *Chebyshev truncated series* thanks to our generic setting.

Chapter 5

Hensel Lifting for Integral-Roots Certificates

5.1 Introduction

Hensel's lemma is a very classical mathematical tool which, given a polynomial P over the integers, and starting from a solution to $P(x) = 0 \bmod p$, constructs a solution to $P(x) = 0 \bmod p^k$ for any k, under mild hypotheses (for instance, that p does not divide the discriminant of P). This lemma was given its final shape by Hensel, in his study of p-adic numbers, where it plays a central role [77]. The algorithm derived from the proof of Hensel's lemma is usually called Hensel lifting.

The strengths of Hensel's lemma are its versatility (it can be stated in a very general context, being also valid, e.g., for power series rings), the fact that it combines an assertion of existence (there is a lifting) and uniqueness (this lifting is unique) under mild hypotheses, and the fact that it is effective, and provides a simple and efficient algorithm.

Hensel's lemma has applications in several areas of mathematics and computer science where one has to solve equations over the integers. In this work, we are interested in some of them that we present just below.

5.1.1 Hensel's Lemma in Computer Algebra

Since the beginning of the development of computer algebra, Hensel's lemma (the algorithmic version of this lemma being usually called Hensel lifting) has been one of the fundamental tools for computations over the integers. The strategy is to isolate a "nice" prime p where the problem under study over the integers has a "good reduction" to the same (but usually easier) problem modulo p^k for all k. One then solves the problem mod p, lifts the solution modulo p^t for some t such that p^t is slightly larger than an *a priori* bound on the size of the solution.

Standard applications include linear system solving over the integers (Hensel's technique is very powerful in practice as it avoids manipulating huge rational numbers while keeping the size of integers involved under control), but also 0-dimensional polynomial system solving over the integers or polynomial factoring over the integers. In all those cases, Hensel's lemma can be used in two ways:

- to compute the solutions;
- to provide a certificate for a list of solutions (using the uniqueness statement of the lemma).

In the present work, we are mainly interested in the second aspect, in a setting where Hensel lifting is used at the end of a complex algorithmic chain where certification is a major issue.

5.1.2 A Certificate-Based Approach for Solving the TMD

The main steps of the SLZ algorithm were recalled in Section 2.5.3. In this algorithmic chain, step 4 is, for standard values of the parameters, in practice by far the most time consuming part of the overall algorithm, due to the call to the LLL algorithm [104]. This suggests to treat the LLL calls as an oracle and log its results in a certificate, the verification of which no longer implying LLL calls. This has the clear benefit of avoiding to have to deal with LLL formally. Still, this approach has to be validated, we need to design "good" certificates that are of reasonable size and can be easily checked, with the goal to provide a fully verified checker for these certificates, in a similar way to what was done for primality certificates in [69, 68, 163].

Roughly speaking, one such certificate will have to gather the coefficients of the linear combinations v_1 and v_2 that appear in step 4, as well as the solutions claimed for step 5, which is where Hensel lifting comes into play, and the core motivation for this chapter.

Note that the part of this work which, beyond Hensel's lemma, addresses Coppersmith's technique might have some use in applications of the latter, for instance to the list decoding of some error-correcting codes [23, 9]

5.1.3 Our Contributions

We formalize Hensel's lemma in the Coq proof assistant for both univariate and bivariate polynomials on $\mathbb{Z}$. Relying on these uniqueness results, we propose some "*small-integral-roots certificates checkers*" that we formally prove correct. Then we implement Coppersmith's technique in the form of a certificate checker for the integer-small-value-problem, this third checker being built upon our *bivariate* small-integral-roots certificates checker. Finally we implement a computational version of these checkers in Coq. This leads to a computational, formal component to be involved in a complete verification chain for solving the Table Maker's Dilemma, based on the SLZ algorithm (see Section 2.5.3).

5.1.4 Outline

The chapter is organized as follow:

- In Section 5.2, we present the Hensel lifting algorithm and we highlight its usefulness to find the small integral roots of polynomials with integer coefficients, then we recall the version of Coppersmith's technique at stake;
- Section 5.3 is devoted to the Coq formalization of Hensel's lemma itself, namely a description of the Coq mechanized formal background that is

common to both univariate and bivariate cases, followed by some pen-and-paper proofs for the uniqueness results that we develop;

- In Section 5.4, we present the formalization of our certificates checkers for the "*univariate and bivariate small-integral-roots problems,*" as well as for the *integer-small-value problem*, and we present the extra formal material that makes it possible to obtain effective checkers in a modular way, despite the non-computational nature of most of the datatypes used in the proofs. We also describe some numerical examples that are typical for the main application that we target;
- In Section 5.5, we discuss the relevance of our formalization choices and we mention the few optimizations that we have implemented in order to increase the efficiency of our effective checkers;
- Finally we draw some conclusions in Section 5.6.

We recall that a comprehensive list of mathematical notations used in the sequel is available in Appendix A.

5.2 Presentation of Hensel Lifting and Coppersmith's Technique

The overall goal of the formalization at stake is to provide certificates checkers based on bivariate Hensel lifting. We start by giving a mathematical and algorithmic description of the univariate version of Hensel lifting to highlight the key ideas behind this technique, and then turn to the bivariate case.

5.2.1 An Overview of Hensel Lifting in the Univariate Case

Before focusing on the uniqueness statement of Hensel's lemma (Lemma 5.2) that constitutes a key part of the present chapter, we start by presenting the algorithm of Hensel lifting with some remarks of an algorithmic nature that are useful to give more intuition on its semantics.

Suppose we know a modular root (mod p) of a given polynomial $P \in \mathbb{Z}[X]$.

The idea is to use a kind of Newton iteration to "lift" this modular root and obtain corresponding roots modulo powers of p, as summarized in Algorithm 5.1.

Algorithm 5.1: Hensel lifting

Input: $P \in \mathbb{Z}[X],\ p \in \mathbb{P},\ u_k \in \mathbb{Z}$ s.t. $P(u_k) \equiv 0 \pmod{p^{2^k}}$ and $P'(u_k) \not\equiv 0 \pmod{p}$.

Output: $u_{k+1} \in \mathbb{Z}$ s.t. $P(u_{k+1}) \equiv 0 \pmod{p^{2^{k+1}}}$.

$$u_{k+1} \leftarrow u_k - \frac{P(u_k)}{P'(u_k)} \bmod p^{2^{k+1}}$$

Note that this corresponds to the typical, *quadratic* version of Hensel lifting as described in [60]—the modulus is *squared* at each iteration—, while some works in computer arithmetic [94, 140] rely on the *linear* variant of Hensel lifting, for prime $p = 2$: they consider moduli of the form $2^i,\ i \in \mathbb{N}^*$.

So we can start with $k := 0$ and iterate Algorithm 5.1 with increasing values of k, which leads to moduli of the form p^{2^k}. Yet we need to make sure that the successive values of u_k satisfy the invertibility assumption modulo p, that is

$$P'(u_k) \not\equiv 0 \pmod{p} \tag{5.1}$$

for each considered u_k. Nevertheless, it is sufficient to assume that the polynomial $P \in \mathbb{Z}[X]$, together with the considered prime p, satisfies the following hypothesis:

$$\forall z \in \mathbb{Z}, \quad P(z) \equiv 0 \pmod{p} \implies P'(z) \not\equiv 0 \pmod{p}, \tag{5.2}$$

this assertion being equivalent to

$$\forall z \in [\![0, p[\![, \quad P(z) \equiv 0 \pmod{p} \implies P'(z) \not\equiv 0 \pmod{p}, \tag{5.3}$$

which can easily be checked in practice (i.e., we need to check only p values for z). We thus obtain *modular roots* modulo an arbitrary big integer $M = p^{2^k}$. Consequently, we can obtain the *small integral roots* of the considered polynomial, as summarized in Algorithm 5.2.

Algorithm 5.2: Find the *small* integral roots of a polynomial using Hensel lifting

Input: $P \in \mathbb{Z}[X]$, $B \in \mathbb{N}^*$.
Output: The integral roots of P whose absolute value is $\leqslant B$.

$p \leftarrow$ a prime (say, the smallest one) such that (5.3) is fulfilled
$S \leftarrow \varnothing$
foreach $z \in [\![0, p[\![$ *such that* $P(z) \equiv 0 \pmod{p}$ **do**
  $k \leftarrow 0$
  $M \leftarrow p$
  $u \leftarrow z$
  // We have $M = p^{2^k}$, $u \equiv z \pmod{p}$, $P(u) \equiv 0 \pmod{M}$ and $0 \leqslant u < M$.
  while $M \leqslant 2 \cdot B$ **do**
    $k \leftarrow k + 1$
    $M \leftarrow M^2$
    $u \leftarrow u - \dfrac{P(u)}{P'(u)} \bmod M$
  end
  // We have $M = p^{2^k}$, $u \equiv z \pmod{p}$, $P(u) \equiv 0 \pmod{M}$ and $0 \leqslant u < M$.
  if $u \leqslant M/2$ **then** $s \leftarrow u$ **else** $s \leftarrow u - M$
  // We have $M = p^{2^k}$, $s \equiv z \pmod{p}$, $P(s) \equiv 0 \pmod{M}$ and $-\frac{M}{2} < s \leqslant \frac{M}{2}$.
  if $P(s) = 0$ **and** $|s| \leqslant B$ **then** $S \leftarrow S \cup \{s\}$
end
return S

Here we can formulate several key remarks related to this algorithm:

Remark 5.1 (Choice of B). On the one hand, Algorithm 5.2 addresses what we can call the *univariate small-integral-roots problem*, namely finding the integral roots s of P such that $|s| \leqslant B$, for a given $P \in \mathbb{Z}[X]$ and $B \in \mathbb{N}^*$. On

the other hand, there exist some results that allow one to bound all the roots of such a univariate polynomial [154]. For instance, Lemma 5.1, which will be stated later on in the present section, gives a result of this kind in the case of *polynomials with integer coefficients*. Consequently, Algorithm 5.2 together with the bound given by Lemma 5.1 can be extended to address the "*univariate whole-integral-roots problem*", that is finding all the integral roots of a given $P \in \mathbb{Z}[X]$, provided (5.3) is satisfied. (We elaborate on this latter issue in Remark 5.2.)

Remark 5.2 (Choice of p). The prime p plays a central role in Hensel lifting, since all computations are carried out modulo a power of p. As mentioned in Algorithm 5.2/Line 1, p has to be chosen such that (5.3) is satisfied, which intuitively amounts to assuming the polynomial P has no repeated roots modulo p. It can be shown that such a p can be found provided that the polynomial P has no multiple root in $\mathbb{Z}$ (which can be accomplished, if need be, by considering $\frac{P(X)}{\gcd\left(P(X),P'(X)\right)}$ instead of $P(X)$). But actually, we need not prove the existence of a suitable prime p in our approach, since we want to focus on the verification of instances of the problem at stake, where the prime p will have been generated by a kind of oracle.

Remark 5.3 (Modular inversion). The modular inversion on Line 10 is the key step of Algorithm 5.2 and is written with a slight abuse of notation. More formally, it could have been written just as follows:

$$u \leftarrow \Big(u - \big(P'(u)\big)_M^{-1} \times P(u)\Big) \bmod M. \tag{5.4}$$

For the sake of completeness, note that it is possible to compute this quantity u at each step with a formula that does not require the modular inversion operation, except for initializing the first step, by computing an inverse modulo p. Such a formula can be found in [60, Algorithm 9.22, p. 264].

Remark 5.4 (Centered modulo). The last-but-one step of Algorithm 5.2 amounts to computing a *centered modulo,* i.e., the instructions on Line 12 are here equivalent to:

$$s \leftarrow u \operatorname{cmod} M. \tag{5.5}$$

However this kind of modulo operation is not directly available in usual implementations of integer arithmetic. If it were the case, since the behavior of Hensel lifting does not depend on the chosen representatives, we could just as well remove Line 12 and directly use a centered modulo on Line 10:

$$u \leftarrow u - \frac{P(u)}{P'(u)} \operatorname{cmod} M. \tag{5.6}$$

A Simple Bound on the Univariate Integral Roots

For any univariate polynomial with *integer* coefficients, the following lemma gives a simple example of a bound on its integral roots. Other bounds might be sharper in some cases, but given the efficiency of Hensel's lemma (which requires a number of steps logarithmic in the bound) the sharpness of the bound used has little significance.

Lemma 5.1. *For any $P \in \mathbb{Z}[X]$, if we write P in the form $\sum_{i=\nu}^{d} a_i X^i \in \mathbb{Z}[X]$ with $a_\nu \neq 0$, we have $\forall z \in \mathbb{Z},\ P(z) = 0 \Rightarrow |z| \leqslant |a_\nu|$. Therefore $B := |a_\nu| = \left|a_{\min\{k\,:\,a_k \neq 0\}}\right|$ is a bound on the integral roots of P.*

Proof. The idea of the proof is as follows: if z is a nonzero integral root of P, we can rewrite $P(z) = 0$ as $a_\nu = -z\,(a_{\nu+1} + \cdots)$, which implies $z \mid a_\nu \neq 0$, hence $|z| \leqslant |a_\nu|$. □

Consequently, we can take the generic bound given by the previous simple lemma if we want to address the *whole-integral-roots problem*. Otherwise we can just choose another bound, depending on the considered application involving the *small-integral-roots problem*.

Example of Use

Let us consider the polynomial

$$P := 225X^5 - 11595X^4 + 9961X^3 - 197931X^2 + 104312X - 13872 \in \mathbb{Z}[X].$$

Suppose we want to determine all the integral roots x of P. By Lemma 5.1, their absolute value satisfies $|x| \leqslant 13872$. We notice that the derivative of P is

$$P' = 1125X^4 - 46380X^3 + 29883X^2 - 395862X + 104312 \in \mathbb{Z}[X].$$

and that the smallest prime satisfying Hensel's lemma hypothesis (5.8) is $p = 3$. Indeed, for $p = 2$, we have

$$\begin{aligned} P(0) &= -13872 \equiv 0 \pmod 2 & \qquad P'(0) &= 104312 \equiv 0 \pmod 2 \\ P(1) &= -108900 \equiv 0 \pmod 2 & \qquad P'(1) &= -306922 \equiv 0 \pmod 2 \end{aligned}$$

(i.e., the derivative cancels on at least one root modulo 2), while for $p = 3$, we have:

$$\begin{aligned} P(0) &= -13872 \equiv 0 \pmod 3 & \qquad P'(0) &= 104312 \not\equiv 0 \pmod 3 \\ P(1) &= -108900 \equiv 0 \pmod 3 & \qquad P'(1) &= -306922 \not\equiv 0 \pmod 3 \\ P(2) &= -695604 \equiv 0 \pmod 3 & \qquad P'(2) &= -920920 \not\equiv 0 \pmod 3. \end{aligned}$$

For each of the roots of P modulo 3, we perform $k = 4$ iterations of Hensel lifting following Algorithm 5.2, given that $3^{2^4} = 43046721 > 2 \times 13872$:

For $z = 0$: the values taken by u cmod M are $0, -3, -30,\ \ 51,$ and $\ 51$.

For $z = 1$: the values taken by u cmod M are $1,\ \ 1,\ -8, -413,$ and $\ 5523949$.

For $z = 2$: the values taken by u cmod M are $2, -1,\ \ 8,\ \ 413,$ and -5523949.

Now we focus on the values computed for the last iterations, and we notice that

$$P(51) = 0 \ \wedge\ |51| \leqslant 13872 \qquad P(5523949) \neq 0 \qquad P(-5523949) \neq 0. \qquad (5.7)$$

We have thus found a small *integral* root of P, 51, among the *modular* roots of P obtained at the last iteration of Hensel lifting. Thanks to the material presented throughout this chapter, we will be able to formally prove that, for example, 51 is the unique integral root of the polynomial P over $\mathbb{Z}$.

Remark 5.5 (Order of multiplicity). Finally, note that if the main assumption (5.3) implies there are no repeated roots in $\mathbb{Z}$, it does not prevent the polynomial from having "repeated roots outside $\mathbb{Z}$," notably in $\mathbb{Q} \setminus \mathbb{Z}$. This is illustrated by the previous example: the polynomial P can be factored to $P = (X - 51)(15X - 4)^2(X^2 + 17)$, implying that $x = \frac{4}{15}$ is a rational root of P with an order of multiplicity greater than 1. However we needed not pre-simplify the polynomial P using the technique mentioned in Remark 5.2 (i.e., dividing P by $\gcd(P, P')$).

A Uniqueness Property on the Modular Roots of $P \in \mathbb{Z}[X]$

We now present the main lemma involved in our formalization of univariate Hensel lifting:

Lemma 5.2 (Hensel, univariate case). *Let $P \in \mathbb{Z}[X]$ and $p \in \mathbb{P}$ that satisfies*

$$\forall z \in [\![0, p[\![\,, \quad P(z) \equiv 0 \pmod{p} \implies P'(z) \not\equiv 0 \pmod{p}, \tag{5.8}$$

where P' is the derivative of the polynomial P. If $x \in \mathbb{Z}$ is such that

$$P(x) \equiv 0 \pmod{p^{2^m}} \tag{5.9}$$

for a given $m \in \mathbb{N}$, then for

$$u_0 := x \bmod p, \tag{5.10}$$

the sequence (u_k) defined by the recurrence relation

$$\forall k \in [\![0, m[\![\quad u_{k+1} := u_k - \frac{P(u_k)}{P'(u_k)} \bmod p^{2^{k+1}} \tag{5.11}$$

satisfies:

$$\forall k \in [\![0, m]\!]\,, \quad u_k = x \bmod p^{2^k}. \tag{5.12}$$

Remark 5.6 (A uniqueness result). Note that Lemma 5.2 gives a necessary condition on each root of $P \in \mathbb{Z}[X]$ modulo p^{2^k} depending on the value of the considered root modulo p. It is somewhat a uniqueness result, whereas usual results about Hensel lifting such as the correctness theorem we can find in [60, p. 264] are existence results. We will see in Sections 5.4.1 and 5.4.2 that we specifically need such a uniqueness property to prove our main theorems that deal with integral-roots certificates.

Remark 5.7 (Inequalities). The technique described in Algorithm 5.2 first considers the roots modulo p, and outputs *as many* roots modulo p^{2^k}, say $\ell \in \mathbb{N}$ such roots, which are candidate for being integral roots (in $\mathbb{Z}$). Consequently, if we denote by r the number of actual integral roots, we obviously have:

$$0 \leqslant r \leqslant \ell \leqslant p. \tag{5.13}$$

This implies that p has to be greater than or equal to the number of actual roots in $\mathbb{Z}$. Otherwise, Hypothesis (5.8) will not be fulfilled.

Before giving a complete mathematical proof of Lemma 5.2 and for illustrating its usefulness, we will present a correctness proof of Algorithm 5.2 as a corollary of Lemma 5.2.

A Correctness Proof for Algorithm 5.2

First, we need to introduce a key arithmetic result, which we will reuse later on in Sections 5.4.1 and 5.4.2:

Lemma 5.3. *For all $M, m, n \in \mathbb{Z}$, if $m \equiv n \pmod{M}$, $|m| \leqslant M/2$ and $|n| < M/2$, then we have $m = n$.*

Proof. Suppose $M, m, n \in \mathbb{Z}$ satisfy the hypotheses of Lemma 5.3, so that we have $|2 \cdot m| \leqslant M$ and $|2 \cdot n| < M$. Using the triangle inequality leads to:

$$|2 \cdot (m - n)| < M + M,$$

hence

$$0 \leqslant |m - n| < M. \qquad \square$$

Since we have also $m - n \equiv 0 \pmod{M}$ by hypothesis, we deduce that $m = n$.

Lemma 5.4 (Correctness of Algorithm 5.2). *For any bound $B \in \mathbb{N}^*$ and for any $P \in \mathbb{Z}[X]$ satisfying (5.3) (same as (5.8)), the finite sets S (returned by the algorithm) and $R := \{\alpha \in \mathbb{Z} \,|\, P(\alpha) = 0 \ \wedge \ |\alpha| \leqslant B\}$ are equal.*

Proof. We proceed by double-inclusion:

- First, we have $S \subset R$ since S is initially empty, and each inserted element on Line 13 does belong to R;
- Next, let $x \in \mathbb{Z}$ be an element of R. Note that $P(x) = 0$ implies

$$P(x \bmod p) \equiv 0 \pmod{p}. \tag{5.14}$$

So the algorithm will process the **foreach** loop for $z := x \bmod p$, then iterate Hensel lifting, until the stop condition $M > 2 \cdot B$ holds. Then we have the following post-condition:

$$\begin{cases} \exists k \in \mathbb{N}, \ M = p^{2^k}, \\ u \equiv z \pmod{p}, \\ 0 \leqslant u < M, \end{cases}$$

where the current value of variable u involved in Algorithm 5.2 corresponds exactly to the quantity u_k involved in Lemma 5.2, and the initial value of u satisfies $u_0 = x \bmod p$. So we can apply this lemma, hence:

$$\begin{cases} u_k \equiv x \pmod{M}, \\ s := u_k \operatorname{cmod} M, \\ s \equiv x \pmod{M}, \\ -\frac{M}{2} < s \leqslant \frac{M}{2}, \\ |x| \leqslant B < \frac{M}{2}, \end{cases}$$

implying that $s = x$ thanks to Lemma 5.3. To sum up, Algorithm 5.2 correctly found the small integral root x, and added it to S. $\square$

Pen-and-Paper Proof of Lemma 5.2

For the sake of completeness, we present the pen-and-paper proof that helped us to carry out the Coq formalization of Lemma 5.2.

Let us assume all the hypotheses of the lemma hold for $P \in \mathbb{Z}[X]$, $p \in \mathbb{P}$, $x \in \mathbb{Z}$, and (u_k) which is defined by (5.10) and (5.11). To shorten most of the following formulas, we will denote $x \bmod p^{2^k}$ by x_k for all integer $k \in \mathbb{N}$.

We want to show (5.12), that is to say, $\forall k \in \mathbb{N},\ k \leqslant m \Rightarrow u_k = x_k$. We prove it by induction on k:

• First, we can notice that $p = p^{2^0}$, therefore (5.10) means that $u_0 = x_0$, which proves the base case.

• The inductive case amounts to showing that for a given integer $k < m$ satisfying $u_k = x_k$, we have $u_{k+1} = x_{k+1}$.

To start with, we can write $x_k = x \bmod p^{2^k} = \left[x \bmod p^{2^{k+1}}\right] \bmod p^{2^k} = x_{k+1} \bmod p^{2^k}$, which implies $x_{k+1} \equiv x_k \pmod{p^{2^k}}$, hence

$$\exists \lambda \in \mathbb{Z}, \quad x_{k+1} = x_k + \lambda p^{2^k}.$$

Now we introduce the operator

$$\Delta_k(P) := \frac{P^{(k)}}{k!} \tag{5.15}$$

and we notice that it maps $\mathbb{Z}[X]$ inside $\mathbb{Z}[X]$ (we can verify it easily on the monomials, which form a basis for $\mathbb{Z}[X]$, and this result actually amounts to saying that the binomial coefficients are integers).
Then we invoke Taylor's theorem, written in terms of Δ, for the polynomial P:

$$\begin{aligned} P(x_{k+1}) &= \sum_{j=0}^{\deg P} \left(\lambda p^{2^k}\right)^j \Delta_j(P)(x_k) \\ &= P(x_k) + \lambda p^{2^k} P'(x_k) + \sum_{j=2}^{\deg P} \left(\lambda p^{2^k}\right)^j \Delta_j(P)(x_k). \end{aligned} \tag{5.16}$$

The left-hand side of (5.16) is zero modulo $p^{2^{k+1}}$, since

$$P(x_{k+1}) = P(x \bmod p^{2^{k+1}}) \equiv P(x) = 0 \pmod{p^{2^{k+1}}}.$$

As for the right-hand side of (5.16), we can notice that

$$\forall j \geqslant 2,\ \left(\lambda p^{2^k}\right)^j = \left[\lambda^j \cdot \left(p^{2^k}\right)^{j-2}\right] \cdot p^{2^{k+1}} \text{ and } \Delta_j(P)(x_k) \in \mathbb{Z}.$$

Thus all terms in the summation involved in (5.16) are zero modulo $p^{2^{k+1}}$ whenever $j \geqslant 2$. Consequently, (5.16) becomes:

$$0 \equiv P(x_k) + \lambda p^{2^k} P'(x_k) + 0 \pmod{p^{2^{k+1}}}. \tag{5.17}$$

Furthermore, we have $P(x_k) \equiv 0 \pmod p$, hence by (5.8), $P'(x_k) \not\equiv 0 \pmod p$, and consequently $P'(x_k)$ is prime to $p^{2^{k+1}}$, hence invertible modulo

$p^{2^{k+1}}$, which allows us to write:

$$\lambda p^{2^k} \equiv -\frac{P(x_k)}{P'(x_k)} \pmod{p^{2^{k+1}}},$$

Now replacing λp^{2^k} with $x_{k+1} - x_k$ and using the induction hypothesis leads to

$$x_{k+1} \equiv u_k - \frac{P(u_k)}{P'(u_k)} \pmod{p^{2^{k+1}}}.$$

Then we recognize the definition (5.11), which means that we have proved that

$$x_{k+1} \bmod p^{2^{k+1}} = u_{k+1},$$

that is, $x_{k+1} = u_{k+1}$. □

This ends the proof of Lemma 5.2 for univariate Hensel lifting. We will present a similar kind of result for bivariate Hensel lifting in the sequel.

5.2.2 Focus on Hensel Lifting in the Bivariate Case

This section is devoted to *bivariate Hensel lifting,* and relies on the generalization of Algorithms 5.1 and 5.2 to bivariate polynomials. We will thus present the generalized algorithms, describing some of their features compared to the univariate case, then give the bivariate version of Lemma 5.2 with a complete pen-and-paper proof.

Generalization of Algorithms 5.1 and 5.2 to Bivariate Polynomials

The definition of bivariate Hensel lifting is very similar to the one of univariate Hensel lifting. A first major change is that we focus on the simultaneous roots of a pair (P_1, P_2) of bivariate polynomials. To be more precise, given $(P_1, P_2) \in \big(\mathbb{Z}[X,Y]\big)^2$, we will be able to consider either its *bivariate integral roots* (i.e., the $(x,y) \in \mathbb{Z}^2$ s.t. $P_1(x,y) = 0 = P_2(x,y)$), or its *bivariate modular roots modulo* M (i.e., the $(x,y) \in [\![0, M[\![^2$ s.t. $P_1(x,y) \equiv 0 \equiv P_2(x,y) \pmod{M}$).

Furthermore, the univariate version of Hensel lifting was involving the derivative $P'(a)$ of the polynomial $P \in \mathbb{Z}[X]$ at some $a \in \mathbb{Z}$, while here we have to consider $J_{P_1,P_2}(a,b)$, the Jacobian matrix[1] of the pair of polynomials $(P_1, P_2) \in \big(\mathbb{Z}[X,Y]\big)^2$ evaluated at $(a,b) \in \mathbb{Z}^2$. This is a natural extension, as the Jacobian matrix (or the differential, which amounts to the same) is the generalization of the derivative in higher-dimensional calculus. Hence the generalized Algorithm 5.3.

Consequently, the main assumption on the roots modulo p that dealt with a nonzero derivative modulo p for univariate polynomials (cf. (5.2)) becomes:

$$\forall z, w \in \mathbb{Z}, \quad P_1(z,w) \equiv 0 \equiv P_2(z,w) \pmod{p} \implies \det J_{P_1,P_2}(z,w) \not\equiv 0 \pmod{p}, \quad (5.18)$$

[1] Recall that this matrix is defined by:

$$J_{P_1,P_2}(a,b) = \begin{pmatrix} \frac{\partial P_1}{\partial X}(a,b) & \frac{\partial P_1}{\partial Y}(a,b) \\ \frac{\partial P_2}{\partial X}(a,b) & \frac{\partial P_2}{\partial Y}(a,b) \end{pmatrix} \in \mathcal{M}_2(\mathbb{Z}).$$

Algorithm 5.3: Bivariate Hensel lifting

Input: $P_1, P_2 \in \mathbb{Z}[X,Y]$, $p \in \mathbb{P}$, $u_k, v_k \in \mathbb{Z}$ s.t. $P_i(u_k, v_k) \equiv 0 \pmod{p^{2^k}}$, $i = 1, 2$, and $\det J_{P_1,P_2}(u_k, v_k) \not\equiv 0 \pmod{p}$.

Output: $(u_{k+1}, v_{k+1}) \in \mathbb{Z}^2$ s.t. $P_i(u_{k+1}, v_{k+1}) \equiv 0 \pmod{p^{2^{k+1}}}, i = 1, 2$.

$$\begin{pmatrix} u_{k+1} \\ v_{k+1} \end{pmatrix} \leftarrow \begin{pmatrix} u_k \\ v_k \end{pmatrix} - \left[J_{P_1,P_2}(u_k, v_k) \right]^{-1}_{p^{2^{k+1}}} \begin{pmatrix} P_1(u_k, v_k) \\ P_2(u_k, v_k) \end{pmatrix} \bmod p^{2^{k+1}}$$

which deals with the invertibility of the matrix $J_{P_1,P_2}(z,w)$ modulo p. Notice that Condition (5.18) can be decided with a finite number of calculations: we only need to check p values for z and p values for w, that is, p^2 different pairs $(z,w) \in [\![0,p[\![^2$. We thus obtain Algorithm 5.4.

Algorithm 5.4: Find the small integral roots of two bivariate polynomials

Input: $P_1, P_2 \in \mathbb{Z}[X,Y]$, $A, B \in \mathbb{N}^*$.
Output: The bivariate integral roots (s,t) of (P_1, P_2) such that $|s| \leqslant A$ and $|t| \leqslant B$.

```
p ← a prime (say, the smallest one) such that (5.18) is fulfilled
S ← ∅
foreach (z, w) ∈ [[0, p[[^2 such that P1(z, w) ≡ 0 ≡ P2(z, w) (mod p) do
    k ← 0
    M ← p
    u ← z
    v ← w
    // Here, M = p^(2^k), u ≡ z (mod p), v ≡ w (mod p), 0 ⩽ u < M, 0 ⩽ v < M.
    while M ⩽ 2 · max(A, B) do
        k ← k + 1
        M ← M^2
        (u, v)^T ← (u, v)^T − [J_{P1,P2}(u, v)]^{-1}_M (P1(u, v), P2(u, v))^T mod M
    end
    // Here, M = p^(2^k), u ≡ z (mod p), v ≡ w (mod p), 0 ⩽ u < M, 0 ⩽ v < M.
    if u ⩽ M/2 then s ← u else s ← u − M
    if v ⩽ M/2 then t ← v else t ← v − M
    // Here, M = p^(2^k), s ≡ z (mod p), t ≡ w (mod p), −M/2 < s ⩽ M/2, −M/2 < t ⩽ M/2.
    if P1(s, t) = 0 = P2(s, t) and |s| ⩽ A and |t| ⩽ B then S ← S ∪ {(s, t)}
end
return S
```

A Uniqueness Property on the Modular Roots of 2 Bivariate Polynomials on $\mathbb{Z}$

An important result which constitutes the key point of our formalization of bivariate Hensel lifting is given by the following lemma, which is a generalization of Lemma 5.2 to the bivariate case.

Lemma 5.5 (Hensel, bivariate case). *Let P_1, P_2 be two bivariate polynomials with integer coefficients, and let p be a prime that satisfies:*

$$\forall z,t \in \mathbb{Z}, \quad P_1(z,t) \equiv 0 \equiv P_2(z,t) \pmod{p} \implies \det J_{P_1,P_2}(z,t) \not\equiv 0 \pmod{p}. \tag{5.19}$$

If $(x,y) \in \mathbb{Z}^2$ is such that

$$P_1(x,y) \equiv P_2(x,y) \equiv 0 \pmod{p^{2^m}} \tag{5.20}$$

for a given $m \in \mathbb{N}$, then for

$$\begin{pmatrix} u_0 \\ v_0 \end{pmatrix} := \begin{pmatrix} x \bmod p \\ y \bmod p \end{pmatrix}, \tag{5.21}$$

the sequence $(u_k, v_k)_k$ defined by the recurrence relation

$$\forall k \in [\![0, m[\![\,, \quad \begin{pmatrix} u_{k+1} \\ v_{k+1} \end{pmatrix} := \begin{pmatrix} u_k \\ v_k \end{pmatrix} - \Big[J_{P_1,P_2}(u_k, v_k) \Big]^{-1}_{p^{2^{k+1}}} \begin{pmatrix} P_1(u_k, v_k) \\ P_2(u_k, v_k) \end{pmatrix} \bmod p^{2^{k+1}} \tag{5.22}$$

satisfies:

$$\forall k \in [\![0, m]\!]\,, \quad \begin{pmatrix} u_k \\ v_k \end{pmatrix} = \begin{pmatrix} x \bmod p^{2^k} \\ y \bmod p^{2^k} \end{pmatrix}. \tag{5.23}$$

Remark 5.6 is also applicable in the context of Lemma 5.5, while Remark 5.7 becomes:

Remark 5.8 (Inequalities in the bivariate case). If we denote by ℓ_{biv} the number of *bivariate modular roots* modulo p^{2^k} (this number being regardless of $k \in \mathbb{N}$), and by r_{biv} the number of actual *bivariate integral roots* of (P_1, P_2) in $\mathbb{Z} \times \mathbb{Z}$, we have:

$$0 \leqslant r_{\text{biv}} \leqslant \ell_{\text{biv}} \leqslant p^2. \tag{5.24}$$

Lemma 5.6 (Correctness of Algorithm 5.4). *For any $A, B \in \mathbb{N}^*$ and any $P_1, P_2 \in \mathbb{Z}[X]$ satisfying (5.18), the finite sets $S \subset \mathbb{Z}^2$ (returned by the algorithm) and $R := \left\{ (x,y) \in \mathbb{Z}^2 \,\middle|\, P_1(x,y) = 0 = P_2(x,y) \ \wedge\ |x| \leqslant A \ \wedge\ |y| \leqslant B \right\}$ are equal.*

Proof. The proof of Lemma 5.6 is very similar to the one of Lemma 5.4; it relies on Lemma 5.5, and uses twice Lemma 5.3 (once for each component of the bivariate integral root). □

Pen-and-Paper Proof of Bivariate Lemma 5.5

Now let us present the pen-and-paper proof that helped us to carry out the formalization of Lemma 5.5 for bivariate polynomials in the Coq proof assistant.

Let us assume all the hypotheses of the lemma hold for $P_1, P_2 \in \mathbb{Z}[X,Y]$, $p \in \mathbb{P}$, $x, y \in \mathbb{Z}$, and $(u_k), (v_k)$ which are defined in (5.21) and (5.22) by mutual recurrence.

To shorten most of the following formulas, we consider the sequences (x_k) and (y_k) of the modular residues of the root (x,y):

$$\forall k \in \mathbb{N}, \quad \begin{pmatrix} x_k \\ y_k \end{pmatrix} := \begin{pmatrix} x \bmod p^{2^k} \\ y \bmod p^{2^k} \end{pmatrix}.$$

We want to show (5.23), that is to say, $\forall k \in \mathbb{N},\ k \leqslant m \Rightarrow (u_k, v_k) = (x_k, y_k)$. We prove it by induction on k:

• First, we notice that $p = p^{2^0}$, therefore (5.21) means that $(u_0, v_0) = (x_0, y_0)$, which proves the base case.

• The inductive case amounts to showing that for a given integer $k < m$ satisfying $(u_k, v_k) = (x_k, y_k)$, we have $(u_{k+1}, v_{k+1}) = (x_{k+1}, y_{k+1})$.

We can write $x_k = x \bmod p^{2^k} = [x \bmod p^{2^{k+1}}] \bmod p^{2^k} = x_{k+1} \bmod p^{2^k}$, which implies $x_{k+1} \equiv x_k \pmod{p^{2^k}}$, hence

$$\exists \lambda \in \mathbb{Z}, \quad x_{k+1} = x_k + \lambda p^{2^k}.$$

Likewise, we obtain:

$$\exists \mu \in \mathbb{Z}, \quad y_{k+1} = y_k + \mu p^{2^k}.$$

For any bivariate polynomial P, we consider the operator

$$\Delta_{i,j}(P) := \frac{1}{i!j!}\left(\frac{\partial^{i+j}}{\partial X^i \partial Y^j} P\right) \tag{5.25}$$

which maps $\mathbb{Z}[X, Y]$ inside $\mathbb{Z}[X, Y]$. Then we apply Taylor's theorem to each bivariate polynomial P_l $(l = 1, 2)$:

$$P_l(x_{k+1}, y_{k+1}) = \sum_{i,j \in \mathbb{N}} \left(\lambda p^{2^k}\right)^i \left(\mu p^{2^k}\right)^j \Delta_{i,j}(P_l)(x_k, y_k). \tag{5.26}$$

The left-hand side of (5.26) is zero modulo $p^{2^{k+1}}$, since

$$P_l(x_{k+1}, y_{k+1}) = P_l(x \bmod p^{2^{k+1}}, y \bmod p^{2^{k+1}}) \equiv P_l(x, y) = 0 \pmod{p^{2^{k+1}}}.$$

Concerning the right-hand side, we can notice that

$$\forall i, j \in \mathbb{N}, \quad i + j \geqslant 2 \implies \left(\lambda p^{2^k}\right)^i \left(\mu p^{2^k}\right)^j = \left[\lambda^i \mu^j \left(p^{2^k}\right)^{i+j-2}\right] \cdot p^{2^{k+1}}$$

and

$$\Delta_{i,j}(P_l)(x_k, y_k) \in \mathbb{Z},$$

therefore all terms in the summation involved in (5.26) are zero modulo $p^{2^{k+1}}$ whenever $i + j \geqslant 2$. As a result, (5.26) becomes

$$0 \equiv P_l(x_k, y_k) + \lambda p^{2^k} \partial_X P_l(x_k, y_k) + \mu p^{2^k} \partial_Y P_l(x_k, y_k) + 0 \pmod{p^{2^{k+1}}}. \tag{5.27}$$

Note that combining (5.27) for both values $l = 1, 2$, we obtain the following matrix equation:

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} P_1(x_k, y_k) \\ P_2(x_k, y_k) \end{pmatrix} + \left[J_{P_1,P_2}(x_k, y_k)\right] \begin{pmatrix} \lambda p^{2^k} \\ \mu p^{2^k} \end{pmatrix} \bmod p^{2^{k+1}}. \tag{5.28}$$

where the modulo operation is applied coordinatewise.

Furthermore, we have $P_1(x_k, y_k) \equiv 0 \equiv P_2(x_k, y_k) \pmod{p}$, hence by (5.19), $\det J_{P_1,P_2}(x_k, y_k) \not\equiv 0 \pmod{p}$, implying $\det J_{P_1,P_2}(x_k, y_k) \not\equiv 0 \pmod{p^{2^{k+1}}}$, which allows us to write

$$-\left[J_{P_1,P_2}(x_k, y_k)\right]^{-1}_{p^{2^{k+1}}} \begin{pmatrix} P_1(x_k, y_k) \\ P_2(x_k, y_k) \end{pmatrix} \equiv \begin{pmatrix} \lambda p^{2^k} \\ \mu p^{2^k} \end{pmatrix} \pmod{p^{2^{k+1}}}. \tag{5.29}$$

Then, we use the induction hypothesis $(x_k, y_k) = (u_k, v_k)$ after replacing λp^{2^k} with $x_{k+1} - x_k$ (resp. μp^{2^k} with $y_{k+1} - y_k$), and we obtain

$$\begin{pmatrix} u_k \\ v_k \end{pmatrix} - \left[J_{P_1,P_2}(u_k, v_k)\right]^{-1}_{p^{2^{k+1}}} \begin{pmatrix} P_1(u_k, v_k) \\ P_2(u_k, v_k) \end{pmatrix} \equiv \begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} \pmod{p^{2^{k+1}}}.$$

We eventually recognize the definition (5.22), which means we have proved that

$$\begin{pmatrix} u_{k+1} \\ v_{k+1} \end{pmatrix} = \begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} \bmod p^{2^{k+1}},$$

that is, thanks to the idempotence of the modulo, $(u_{k+1}, v_{k+1}) = (x_{k+1}, y_{k+1})$.

5.2.3 Integer Small Value Problem (ISValP) and Coppersmith's Technique

The overall goal of this work is to address the so-called Integer Small Value Problem (ISValP) [160, Section 3.2] in a formal setting. We give below a mathematical presentation of this problem that is typically solved by using Coppersmith's technique [41] (see also the description of the whole SLZ algorithm given in Section 2.5.3).

Given $P \in \mathbb{Z}[X]$, we want to *find the small integer entries on which the univariate polynomial P has small values modulo a large integer M*.

In other words, we want to find all $e \in \mathbb{Z}, |e| \leqslant B$ such that $|P(e) \bmod M| \leqslant A$ for given $M, A, B \in \mathbb{N}$.

If we pose $a = P(e) \bmod M$, we have $|a| \leqslant A$ and $P(e) - a \equiv 0 \pmod{M}$, so that the ISValP problem reduces to *finding the small modular roots of the bivariate polynomial $Q(a, e) = P(e) - a$ modulo M*.

Using the usual indeterminates X and Y in place of variables a and e leads to the definition $Q(X, Y) = P(Y) - X$, which constitutes a slightly different wording with respect to [160], on which we will elaborate at Section 5.5.1.

Coppersmith's technique relies on the choice of a positive integer $\alpha > 0$, as well as the introduction of the family of polynomials

$$Q_{i,j}(X, Y) = Q^i(X, Y) M^{\alpha - i} Y^j$$

for $i \leqslant \alpha$ and $j \leqslant (\alpha - i) \cdot \deg(P)$.

Let (a, e) be a small modular root of Q modulo M. Then we can prove that (a, e) is a modular root of each $Q_{i,j}(X, Y)$ modulo M^α. Furthermore, let us assume that we have found two linear combinations $v_1(X, Y)$ and $v_2(X, Y)$ over $\mathbb{Z}$ of these polynomials $Q_{i,j}(X, Y)$, such that

$$\forall l \in \{1, 2\}, \quad \forall x, y \in \mathbb{Z}, \quad |x| \leqslant A \ \wedge \ |y| \leqslant B \implies |v_l(x, y)| < M^\alpha. \tag{5.30}$$

For both $l \in \{1, 2\}$, this implies that (a, e) is a *modular* root of v_l modulo M^α, and at the same time that $|v_l(a, e)| < M^\alpha$. Thence, we can conclude that (a, e) is a small *integral* root of both bivariate polynomials v_1 and v_2.

These polynomials v_1 and v_2 can be found by means of the LLL algorithm for Euclidean lattice basis reduction, which is the most time consuming part of the overall algorithm SLZ [160]. More details on the way the polynomials v_1 and v_2 are computed may be found in [158, 160, 159]. Yet as regards formal verification, we can just treat the LLL calls as an oracle and log its results in an appropriate certificate, the verification of which no longer implying LLL calls.

We now turn to the description of our formalization in Coq: the upcoming Section 5.3 will be devoted to the formal proof of Hensel's lemma, while in Section 5.4 we will describe the design of some integral-roots certificates, based on Hensel lifting.

5.3 Formalization of Hensel Lifting

In this section we present the different choices we have made to undertake the formalization[2] inside the Coq proof assistant [10] using the SSReflect extension [64, 65]. In particular, Section 5.3.1 is devoted to our final formalization choices for proving Hensel's lemma, for the univariate case (Section 5.3.2), as well as for the bivariate case (Section 5.3.3).

In the sequel, we will often give the name of the formal definitions and lemmas as they appear in the Coq code, in order to help the reader desirous of immersing oneself in the reading of the formal development.

5.3.1 Formal Background for Hensel Lifting

We start by presenting the different theories on which relies our formalization, as well as our new support results that are common to both univariate and bivariate Hensel lifting.

Note that concerning the basic types, functions and lemmas required for this formalization, we can see from the previous Section 5.2 that we need:

(i) $\mathbb{N}$ with the usual operations including exponentiation, and the primality and divisibility predicates;

(ii) $\mathbb{Z}$ with the usual operations including modulo;

(iii) ExtendedGCD-based modular inversion in $\mathbb{Z}/q\mathbb{Z}$ (with $q = p^{2^k}$);

(iv) $\mathbb{Z}[X]$ with polynomial evaluation, formal derivatives, and Taylor's theorem (for polynomials).

The SSReflect Extension.

As said in Section 3.2.7, SSReflect consists of an extension of the Coq proof language as well as a set of Coq libraries developed upon this extension.

In particular, we extensively used the following theories that supply most of the key concepts involved in our formalization:

[2] The Coq development is available at http://tamadi.gforge.inria.fr/CoqHensel/

ssrnat for boolean comparison predicates on the type `nat` of natural numbers, with the usual operations including exponentiation;

prime for the primality predicate `(prime : nat -> bool)`;

div for the divisibility predicate `(dvdn : nat -> nat -> bool)` as well as the modulo `(modn : nat -> nat -> nat)` on which it is based;

zmodp for the ExtendedGCD-based modular inversion in $\mathbb{Z}/q\mathbb{Z}$ (where q will be instantiated by p^{2^k});

poly for the ring of univariate polynomials, with polynomial evaluation, formal derivatives, and Taylor's theorem for univariate polynomials;

The design of all the SSReflect libraries follow the so-called *small-scale reflection* methodology, in such a way that symbolic representations (including the type `bool`) are ubiquitous. Thus mathematical predicates will typically be formalized as Boolean functions. In particular our Coq formalization takes advantage of the primality predicate "`prime : nat -> bool`" provided by the SSReflect library `prime`: a key feature of this predicate is that its codomain is not the logical sort `Prop` of Coq but the enumerated type `bool`, which is suitable for computations.

As regards polynomials, we needed to prove Taylor's theorem for both univariate and bivariate cases. This latter result is one of the key theorems gathered in our theory `bipoly`.

Polynomials with Integer Coefficients.

As regards the type of relative integers, we use the one provided by the Coq standard library `ZArith`. (We will elaborate on this choice later on in Section 5.5.1.)

As regards the definition of polynomials, both `poly` and `bipoly` theories deal with polynomials with coefficients in a type that has to be equipped with a SSReflect decidable-equality ring structure.

Therefore, to be able to deal with polynomials on `Z`, we first needed to prove that `Z` satisfies the required axioms of the SSReflect algebraic hierarchy. This is accomplished in the first part of our theory `ssrzarith`, where we declare some appropriate *Canonical Structures* (a kind of "structure inference" that is heavily used in SSReflect libraries [59]) for the type `Z`. This includes canonical structures `Z_eqType` (for decidable equality), `Z_idomainType` (for integral domain), and `Z_countType` (for countable type).

Now the terms `{poly Z}` and `{bipoly Z}` typecheck so that we can use them in the sequel to designate univariate and bivariate polynomials on $\mathbb{Z}$.

Handling Different Definitions of the Modular Reduction.

In this section, we will summarize why we need to use several definitions of the modular reduction in our formalization. Then we will present some lemmas that allow one to move from one definition of to another.

To sum up, we are using the following four functions related to the modular reduction:

```
Zmod : Z -> Z -> Z
modn : nat -> nat -> nat
ZtoZp : forall q : nat, Z -> 'Z_q
inZp : forall p' : nat, nat -> 'I_p'.+1
```

The first one, `Zmod`, is directly linked to the problem at stake and is relatively efficient, while the second one, `modn`, is involved in most arithmetic lemmas we are using from the SSReflect library, which are expressed in terms of Peano, unary integers.

We provide a link between `modn` and `Zmod` through the following key result:

```
Lemma Z_of_nat_moduli :
  forall n q : nat, q > 0 ->
  Z_of_nat (modn n q) = Zmod (Z_of_nat n) (Z_of_nat q).
```

Furthermore, for any fixed second argument $q \geqslant 2$, none of these two functions are homomorphisms, since $(m+n) \bmod q \neq (m \bmod q) + (n \bmod q)$ in the general case (unless we add a outermost modulo operation in the right-hand-side). However, to prove some key results such as *the compatibility between the modulo operation and the polynomial evaluation,* on `{poly Z}` as well as `{bipoly Z}`, we need to apply lemmas that expect a morphism as argument.

Consequently, we need to specify the surjective morphism from $\mathbb{Z}$ onto $\mathbb{Z}/q\mathbb{Z}$, even though it is just for proving purposes.

We thus define a function `ZtoZp`, whose codomain `'Z_q` is a SSReflect notation for `'I_(Zp_trunc q).+2`, that is `'I_q.-2.+2`, where `'I_n` is itself a notation for the type "`ordinal n`" defined by:

```
Inductive ordinal (n : nat) : predArgType :=
  | Ordinal : forall m : nat, m < n -> 'I_n.
```

In other words, `'I_q` represents the finite set $[\![0, q[\![$ for $q \in \mathbb{N}$, while `'Z_q` represents the same set $[\![0, q[\![$ with the additional assumption that $q \geqslant 2$. This assumption is syntactically enforced thanks to the term `q.-2.+2` above, which is always $\geqslant 2$. Consequently `'Z_q` becomes a nontrivial ring, which corresponds to the mathematical ring $\mathbb{Z}/q\mathbb{Z}$.

Note that we define `ZtoZp` with the help of the `inZp` function, which is located in the SSReflect library `zmodp` and defined by means of the `modn` function.

Finally we prove that "`ZtoZp q`" is a ring homomorphism for any fixed $q \geqslant 2$:

```
Lemma ZtoZp_morph :
  forall (q : nat) q > 1 -> rmorphism (ZtoZp q).
```

The proof uses the following result linking `ZtoZp` and `Zmod` several times:

```
Lemma ZtoZp_Zmod :
  forall (q : nat) (z : Z), q > 1 ->
  Z_of_nat (ZtoZp q z) = Zmod z (Z_of_nat q).
```

This can be straightforwardly proved, relying on `Z_of_nat_moduli` and on our definition of `ZtoZp`.

Using the previous lemmas, we can now prove some key results such as:

```
Lemma horner_Zmod_compat :
  forall (P : {poly Z}) (z : Z) (q : nat),
  P.[z mod q] = P.[z] %[Zmod q].
```

```
Lemma bieval_Zmod_compat :
  forall (P : {bipoly Z}) (u v : Z) (q : nat), q > 1 ->
  P.2[u mod q, v mod q] = P.2[u, v] %[Zmod q].
```

As regards the notations, "`a mod b`" is the infix notation for "`Zmod a b`" and "`a = b %[Zmod c]`" denotes the equivalence relation $a \equiv b \pmod{c}$, on $\mathbb{Z}$. Note that the SSReflect library `div` provides similar notations for the modulo operation on type `nat`: "`a %% b`" is the infix notation for "`modn a b`" and "`a = b %[mod c]`" is a shortcut for "`(a %% c == b %% c)`". Finally, "`P.[x]`" denotes the Horner evaluation at point `x` of a univariate polynomial `P`, while "`P.2[x,y]`" denotes the evaluation at (x, y) of a bivariate polynomial `P`.

5.3.2 Insights into the Coq Formalization of Univariate Lemma 5.2

Thanks to the material presented in Section 5.3.1, we can define the univariate Hensel lifting as a Coq function:

```
Definition univ_hensel_step (P:{poly Z}) (p:nat) (u:Z) (k:nat) :=
  (u - P.[u] * Zmod_inv (p^2^k.+1) (P^'()).[u]) mod p^2^k.+1.
```

where `P^'()` denotes the derivative of the univariate polynomial `P` and the function `(Zmod_inv : forall q : nat, Z -> 'Z_q)` is the modular inversion that we defined beforehand, by composing the projection `ZtoZp` (from $\mathbb{Z}$ to $\mathbb{Z}/q\mathbb{Z}$), followed by the function `Zp_inv` (from $\mathbb{Z}/q\mathbb{Z}$ to itself). Note that this latter function (provided by the SSReflect library `zmodp`) is a total function that takes two arguments q and x, and returns the desired modular inverse x_q^{-1} if it exists (i.e., if x is coprime with q), otherwise x as a default value.

The function `univ_hensel_step` thus formalizes Algorithm 5.1, and can be iterated by means of a fixpoint, namely:

```
Fixpoint univ_hensel_iterated
  (P : {poly Z}) (p : nat) (u0 : Z) (k : nat) {struct k} : Z :=
  if k is j.+1 then univ_hensel_step (univ_hensel_iterated u0 j) j
  else u0.
```

Now we can state Lemma 5.2 in Coq:

```
Section UnivariateHenselLemma.
Variable P : {poly Z}.
Variable p : nat.
Hypothesis p_prime : prime p.
Hypothesis P_roots_mod_p :
  forall z : Z, 0 <= z < Z_of_nat p ->
  P.[z] = 0 %[Zmod p] -> (P^'()).[z] <> 0 %[Zmod p].
Variable x : Z.
Variable m : nat.
Hypothesis x_root_m : P.[x] = 0 %[Zmod p^2^m].
Let u0 := x mod p.
Let uk := univ_hensel_iterated P p u0.
Let xk k := x mod (p^2^k)%N.
```

```
Lemma univ_hensel_lemma :
  forall k : nat, k <= m -> uk k = xk k.
End UnivariateHenselLemma.
```

We start the proof by induction on the integer k. The base case is solved trivially, while in the inductive case we need to invoke Taylor's theorem for univariate polynomials on a ring `R` (which is not assumed commutative, hence the hypothesis "`GRing.comm x h`" that `x` and `h` commute):

```
Lemma nderiv_taylor :
  forall (R : ringType) (P : {poly R}) (x h : R),
  GRing.comm x h ->
  P.[x + h] = \sum_(i < size P) (P^`N(i)).[x] * h ^+ i.
```

where `P^`N(i)` is a notation for the operator "`nderivn p i`" corresponding to the operator $\Delta_i(P)$ defined in Equation (5.15). Both `nderivn` and `nderiv_taylor` have been been included in the SSReflect library `poly` at the occasion of this work. Note that since SSReflect defines univariate polynomials as lists of coefficients without trailing zeros, we can reuse the `size` function relative to lists, so that for any `P` $\in R[X]$, we have intuitively:

$$\texttt{size P} = \begin{cases} 0 & \text{if } \texttt{P} = 0 \\ 1 + \deg(\texttt{P}) & \text{otherwise.} \end{cases}$$

Following the pen-and-paper proof presented in Section 5.2.1, we need to "truncate" the Taylor expansion given in (5.16) in order to retrieve the "first two terms" involved in (5.17). For this purpose, we first proved the following result that is based on the machinery provided in SSReflect library `bigop` [11].

```
Lemma trunc_univ_sum :
  forall (F : nat -> Z) (n : nat) (q : nat),
  q > 1 ->
  n > 1 ->
  (forall i : nat, 2 <= i < n -> F i = 0 %[Zmod q]) ->
  \sum_(i < n) F i = F 0 + F 1 %[Zmod q].
```

where the notation "`\sum_(i < n) F i`" denotes the finite sum $\sum_{0 \leqslant i < n} F(i)$.

Then we proved that under hypotheses `P_roots_mod_p` and `x_root_m`, we have:

```
Lemma size_P_gt1 : size P > 1.
```

which allows us to prove the side-condition that appears when applying lemma `trunc_univ_sum` (due to the hypothesis "`n > 1`" above).

5.3.3 Insights into the Coq Formalization of Bivariate Lemma 5.5

We now focus on the formalization of bivariate Hensel lifting, which is very similar to the univariate case that we presented in Section 5.3.2. We will thus summarize the main steps we met during the formalization of the bivariate case, highlighting the key differences between both cases.

A first required building block to formalize bivariate Hensel lifting is the availability of some operations on 2-by-2 matrices, including the Cramer rule in modular arithmetic. More precisely, this rule is in charge of computing the quantity $\left[J_{P_1,P_2}(u_k,v_k)\right]^{-1}_{p^{2^{k+1}}} \begin{pmatrix} P_1(u_k,v_k) \\ P_2(u_k,v_k) \end{pmatrix} \bmod p^{2^{k+1}}$ involved in (5.22), and its correctness proof will allow one to deduce (5.29) from (5.28). So in library `morebipolyz`, we define order-2 matrices on a type `T` as a record called `mat2by2` with four projections (one for each matrix coefficient), and we define the functions `Cramer_x` and `Cramer_y` of type:

```
Cramer_x : nat -> mat2by2 Z -> Z -> Z -> Z
Cramer_y : nat -> mat2by2 Z -> Z -> Z -> Z
```

using the following formulas based on order-2 determinants:

$$\forall q \geqslant 2, \quad \forall u, v \in \mathbb{Z}, \quad \forall A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathcal{M}_2(\mathbb{Z}), \begin{cases} \texttt{Cramer\_x}(q, A, u, v) = \begin{vmatrix} u & b \\ v & d \end{vmatrix} \times \begin{vmatrix} a & b \\ c & d \end{vmatrix}^{-1}_q \\ \texttt{Cramer\_y}(q, A, u, v) = \begin{vmatrix} a & u \\ c & v \end{vmatrix} \times \begin{vmatrix} a & b \\ c & d \end{vmatrix}^{-1}_q \end{cases}$$

where the modular inversions can be performed by using the function `Zmod_inv` presented at the beginning of Section 5.3.2. We will motivate our choice for formalizing the concept of 2-by-2 matrices used here later on in Section 5.5.1.

Furthermore, we need to formalize bivariate polynomials on $\mathbb{Z}$. We chose to define bivariate polynomials on a ring `R` by iterating twice the polynomials ring constructor: we introduce the notation `{bipoly R}` as a shortcut for `{poly {poly R}}`, and we instantiate this definition with ring `R := Z`. (We will motivate this choice later on in Section 5.5.1.)

Then, relying on `Cramer_x` and `Cramer_y`, we can implement Algorithm 5.3 as a Coq function `biv_hensel_step`, whose type is as follows:

```
biv_hensel_step :
  {bipoly Z} -> {bipoly Z} -> nat -> Z * Z -> nat -> Z * Z
```

Note that the main difference here with respect to `univ_hensel_step` is that we have two bivariate polynomials instead of a single univariate one, and that the algorithm acts on a pair of integers. Like in the univariate case, we can iterate this first function, which leads to a function `biv_hensel_iterated`, which has the same type:

```
biv_hensel_iterated :
  {bipoly Z} -> {bipoly Z} -> nat -> Z * Z -> nat -> Z * Z
```

Now we can state Lemma 5.5 in Coq:

```
Section BivariateHenselLemma.
Variables P1 P2 : {bipoly Z}.
```

```
Variable p : nat.
Hypothesis p_prime : prime p.
Hypothesis Behavior_roots_mod_p :
  forall (z w : Z),
  0 <= z < p -> 0 <= w < p ->
  P1.2[z,w] = 0 %[Zmod p] ->
  P2.2[z,w] = 0 %[Zmod p] ->
  (Jdet P1 P2).2[z,w] <> 0 %[Zmod p].
Variables x y : Z.
Variable m : nat.
Hypothesis P1_root_m : P1.2[x,y] = 0 %[Zmod p^2^m].
Hypothesis P2_root_m : P2.2[x,y] = 0 %[Zmod p^2^m].
Let x0 := x mod p.
Let y0 := y mod p.
Let hk := biv_hensel_iterated P1 P2 p (x0, y0).
Local Notation xk := (fun k : nat => x mod p^2^k) (only parsing).
Local Notation yk := (fun k : nat => y mod p^2^k) (only parsing).
Lemma biv_hensel_lemma :
  forall k : nat, k <= m -> hk k = (xk k, yk k).
End BivariateHenselLemma.
```

where the expression `(Jdet P1 P2).2[z,w]` corresponds to the determinant of the Jacobian matrix of `P1` and `P2`, evaluated at $(\mathtt{z}, \mathtt{w})$.

Like in the univariate case, we start the proof by induction on the integer k. The base case is solved trivially, while for the inductive case we need Taylor's theorem for bivariate polynomials. We have formalized this latter theorem in our `bipoly` theory.

Furthermore, we need to truncate the Taylor expansion in order to retrieve the "first three terms" involved in (5.27). Yet for this key step we cannot directly use the proof path that worked for the univariate case in Section 5.3.2, since there is no "bivariate equivalent" to the lemma `size_P_gt1`. (This is due to the fact that an *order-2* matrix that is invertible can possibly have some zero coefficients.) We thus have been led to prove the following ad-hoc lemma, that is a little more involved than the lemma `trunc_univ_sum` that we presented in Section 5.3.2:

```
Lemma trunc_biv_sum :
  forall (q : nat) (F : nat -> nat -> Z) (m : nat) (n : nat -> nat),
  q > 1 ->
  ( forall i j : nat, i + j >= 2 -> F i j = 0 %[Zmod q] ) ->
  ( n 1 >= 1 -> m >= 2 ) ->
  ( n 0 >= 1 -> m >= 1 ) ->
  ( n 1 < 1 -> F 1 0 = 0 ) ->
  ( n 0 < 2 -> F 0 1 = 0 ) ->
  ( n 0 < 1 -> F 0 0 = 0 ) ->
  \sum_(i < m)\sum_(j < n i)(F i j) = F 0 0 + F 0 1 + F 1 0 %[Zmod q].
```

The idea here for proving the equality

$$\sum_{0 \leqslant i < m} \sum_{0 \leqslant j < n_i} F_{i,j} = F_{0,0} + F_{0,1} + F_{1,0} \pmod{q}$$

is first to use the hypothesis saying that only terms $F_{i,j}$ with $i+j<2$ may be nonzero modulo q, and at the same time to assert that if ever one such term $F_{i,j}$ with $i+j<2$ is not included in the left-hand side (depending on the values of m and n_i), it will actually be zero. Hence the five implicative hypotheses of `trunc_biv_sum`. We thus prove this lemma by case analysis (18 subgoals), relying on some "sum bookkeeping" that is highly facilitated by the SSReflect library `bigop`.

Finally, we follow the arguments that were presented in a "matrix fashion" at the end of Section 5.2.2, working coefficient by coefficient. This is accomplished with the help of half a dozen lemmas that are linked to Cramer rule for 2-by-2 matrices, including:

```
Lemma Cramer_x_opp : forall q A u v, q > 1 ->
  Cramer_x q A (- u) (- v) = - Cramer_x q A u v.

(* and *)

Lemma Cramer_x_mod : forall q A u v, q > 1 ->
  Cramer_x q A (u mod q) (v mod q) = Cramer_x q A u v %[Zmod q].
```

We will motivate our choice for formalizing the concept of 2-by-2 matrices used here later on in Section 5.5.1.

5.4 Integral Roots Certificates

We follow the skeptical approach by viewing Algorithms 5.2 and 5.3 as certifying algorithms, and devise some "integral root certificates" whose verification need not recompute all the iterations of Hensel lifting. In upcoming Sections 5.4.1 and 5.4.2, we address both univariate and bivariate cases, the latter one being a key building block for the ISValP problem that we will address in Section 5.4.3.

5.4.1 Univariate Case

In the sequel, we will call *univariate small-integral-roots problem* the problem that consists of knowing all the "small" integral roots $x \in \mathbb{Z}$ of a univariate polynomial P on $\mathbb{Z}$. In other words, we consider the following kind of data:

$$\begin{cases} B \in \mathbb{N} \\ P \in \mathbb{Z}[X] \\ \mathcal{S} \subset \mathbb{Z} \end{cases} \tag{5.31}$$

and we want to formally verify that we have

$$\forall x \in \mathbb{Z}, \quad x \in \mathcal{S} \iff \big(|x| \leqslant B \ \wedge \ P(x) = 0\big). \tag{5.32}$$

Following the certificate-based approach described in Section 3.2.6, we first have to define a type for our univariate small-integral-roots certificates. Typically, it should gather at the same time the "input/output" of the problem—here given in Equation (5.31)—, as well as some additional data that allows one to deduce that the considered instance of the problem is valid—here it amounts to saying that Equation (5.32) holds.

We thus consider the following type of certificate for the univariate small-integral-roots problem:

```
Record univCertif := UnivCertif
{ uc_P : {poly Z}
; uc_B : Z
; uc_p : nat
; uc_k : nat
; uc_L : seq (Z * bool)
}.
```

Note that in addition to the polynomial P and the bound B that constitutes the "input" of the problem, we store in the certificate the prime p and the final value of k that both occur in the Algorithm 5.2 of univariate Hensel lifting. More precisely, p is the prime number that will fulfill (5.8), and k is the number of iterations of Hensel lifting performed to reach bound B; in other words, we will have $p^{2^k} > 2 \cdot B$. Finally, the list L involved in the certificate gathers all the roots of P modulo p^{2^k}, along with a Boolean value that indicates whether each *modular root* of the polynomial is an actual *integral root* or not.

Remark 5.9 (Usefulness of the Boolean values). We will see the correctness of the method specifically relies on the fact all the *modular* roots are stored in list L. Yet we want to deal with *integral* roots of the polynomial P, which are obviously among these modular roots. So the presence of the Boolean values inside the elements of list L allows one to easily get all the claimed integral roots of the polynomial from the list L. More precisely, for a given certificate "`uc : univCertif`," we can immediately get the set $\mathcal{S}$ mentioned in (5.31) by considering "`map1_filter2 Z (uc_L uc)`," where `map1_filter2` is defined by:

```
Definition map1_filter2 (T : Type) (L : seq (T * bool)) :=
  map (fun e => e.1) (filter (fun e => e.2) L).
```

We will say that one such certificate (P, B, p, k, L) is valid if the following conditions are fulfilled:

$$p \text{ is a prime number,} \tag{5.33a}$$

$$\text{the integer } k \geqslant 0 \text{ satisfies } p^{2^k} > 2 \cdot B, \tag{5.33b}$$

and denoting $L_p = \{u \bmod p \,|\, \exists b \in \mathbb{B},\ (u, b) \in L\}$, we have:

$$\text{the elements of } L_p \text{ are pairwise distinct,} \tag{5.33c}$$

$$\forall s \in [\![0, p[\![,\ \ s \in L_p \iff P(s) \equiv 0 \pmod{p}, \tag{5.33d}$$

and for all $(u, b) \in L$, we have:

$$P'(u) \not\equiv 0 \pmod{p}, \tag{5.33e}$$

$$|2 \cdot u| \leqslant p^{2^k}, \tag{5.33f}$$

$$P(u) \equiv 0 \pmod{p^{2^k}}, \tag{5.33g}$$

$$b = \mathsf{true} \iff \big(|u| \leqslant B \ \wedge\ P(u) = 0\big). \tag{5.33h}$$

All these Boolean conditions are implemented in the form of a Coq function `univ_check` that has type "`univCertif -> bool`," and which constitutes what we call a *univariate small-integral-roots certificates checker*.

Before dealing with the correctness proof of this checker, we can formulate a few remarks:

Remark 5.10 (Uniqueness predicate). Equation (5.33c) is not central for the problem at stake, yet it can be easily implemented using the `uniq` predicate from the SSReflect library `seq`, and it ensures the number of modular (resp. integral) roots is relevantly described by the size of list L (resp. the size of list $\mathcal{S}$ mentioned in Remark 5.9).

Remark 5.11 (Invertibility hypothesis). Equation (5.33e) can be viewed as a "distributed version" of the main invertibility hypothesis (5.3) for univariate Hensel lifting. Indeed, according to (5.33d), the values that are given to variable u in (5.33e) correspond to all the modular roots of P modulo p, so that (5.33d) and (5.33e) imply

$$\forall u \in [\![0, p[\![\,, \quad P(u) \equiv 0 \pmod{p} \implies P'(u) \not\equiv 0 \pmod{p}.$$

The correctness proof of the univariate checker `univ_check` consists of proving that any certificate that is accepted by the checker is valid, i.e., contains all the integral roots of the considered polynomial in the considered range. To be more precise, we need to prove the following result:

Lemma 5.7 (Correctness of univariate checker). *For all univariate small-integral-roots certificate* `uc` $= (P, B, p, k, L)$ *such that* `(univ_check uc)` *holds, for all* $z \in \mathbb{Z}$ *we have the equivalence*

$$\big(|z| \leqslant B \ \wedge \ P(z) = 0\big) \iff z \in \texttt{(uc\_roots uc)}, \tag{5.34}$$

where `(uc_roots uc)` *is defined by "*`map1_filter2 Z (uc_L uc)`*".*

We now present the main steps of this main correctness proof that we have mechanized in Coq. The reasoning is somewhat close to the proof of Lemma 5.4 that we gave for pedagogical purposes, yet there is a key difference between Lemmas 5.4 and 5.7, namely the former was a usual proof of correctness for what we could call a "certified algorithm", while we now focus on the correctness of a checker, for certificates that are supposed to be generated by a "certifying algorithm", that is a kind of oracle.

Proof. Suppose `(univ_check uc)` hold for a given certificate `uc` $= (P, B, p, k, L)$, and let $z \in \mathbb{Z}$.

- For proving the "$\implies$" part of equivalence (5.34), we will use Lemma 5.2: To begin with, the assumption $P(z) = 0$ implies

  $$P(z \bmod p) \equiv 0 \pmod{p}.$$

  So applying (5.33d) with $s := z \bmod p$ leads to $s \in L_p$, that is by definition of L_p,

  $$\exists (u, b) \in L, \quad s = u \bmod p,$$

hence

$$z \bmod p \equiv u \bmod p. \tag{5.35}$$

Then it is sufficient to show that $(u, b) = (z, \mathsf{true})$, since it implies $(z, \mathsf{true}) \in L$, that is to say, $z \in$ `(uc_roots uc)`.

- First, we want to show that $z = u$. The idea is to apply twice Lemma 5.2 (once for $x := z$ and once for $x := u$). The main hypothesis (5.8) is fulfilled thanks to (5.33e), as explained in Remark 5.11. Moreover, we have immediately

$$P(z) \equiv 0 \pmod{p^{2^k}}$$

and

$$P(u) \equiv 0 \pmod{p^{2^k}}$$

by using (5.33g). So we can apply Lemma 5.2 twice[3] and use (5.35) to deduce that

$$z \bmod p^{2^k} = u \bmod p^{2^k}.$$

In addition, we have

$$\begin{cases} |z| \leqslant B & \text{by hypothesis} \\ B < p^{2^k}/2 & \text{by (5.33b)} \\ |u| \leqslant p^{2^k}/2 & \text{by (5.33f).} \end{cases}$$

Consequently Lemma 5.3 implies $z = u$.

- Second, we want to show that $b = \mathsf{true}$, knowing that (5.33h) holds. It suffices here to combine the hypothesis $\big(|z| \leqslant B \wedge P(z) = 0\big)$ with the fact that $z = u$ (proved just now) to get $\big(|u| \leqslant B \wedge P(u) = 0\big)$, which is equivalent to the desired result $b = \mathsf{true}$.

- For the "$\Longleftarrow$" part of (5.34), we need to verify the values stored in `(uc_roots uc)` are actual integral roots, which is indeed the case since we have

$$z \in \texttt{(uc\_roots uc)} \iff (z, \mathsf{true}) \in L,$$

which implies $\big(|z| \leqslant B \wedge P(z) = 0\big)$ thanks to the condition (5.33h). □

An Example of Univariate Small-Integral-Roots Certificate. Resuming the example of Section 5.2.1 page 90, the calculations based on univariate Hensel lifting allow one to generate the following univariate small-integral-roots certificate, where the Booleans that have to satisfy (5.33h) have been set according to the simple tests mentioned in Equation (5.7).

$$C := \Big(\Big(P := 225X^5 - 11595X^4 + 9961X^3 - 197931X^2 + 104312X - 13872\Big), \Big(B := 13872\Big),$$
$$\Big(p := 3\Big), \Big(k := 4\Big), \Big(L = \Big[(51, \mathsf{true}); (5523949, \mathsf{false}); (-5523949, \mathsf{false})\Big]\Big)\Big).$$

[3] Indeed, in this context we do not know how u has been computed, unlike Lemma 5.4 whose proof only required one application of Lemma 5.2.

Finally, this simple example allows one to formulate a general remark:

Remark 5.12 (Changing p). It is sometimes possible to choose a slightly bigger prime p than the first one satisfying Hypothesis (5.8) so that the list L of the certificate is smaller. (Note that this will typically be useful when there is no root in $\mathbb{Z}$ at all, so that such a change of prime p leads to an empty list L.) In the context of the considered example, we can thus notice that $p = 5$ is another possible prime:

$$\begin{aligned} P(0) &= -13872 \not\equiv 0 \pmod 5 \\ P(1) &= -108900 \equiv 0 \pmod 5 \qquad P'(1) = -306922 \not\equiv 0 \pmod 5 \\ P(2) &= -695604 \not\equiv 0 \pmod 5 \\ P(3) &= -2097888 \not\equiv 0 \pmod 5 \\ P(4) &= -4863936 \not\equiv 0 \pmod 5. \end{aligned}$$

We would thus obtain the following certificate:

$$C' := \Big(\Big(P := 225X^5 - 11595X^4 + 9961X^3 - 197931X^2 + 104312X - 13872\Big), \Big(B := 13872\Big), \Big(p := 5\Big), \Big(k := 3\Big), \Big(L = \Big[(51, \mathsf{true})\Big]\Big)\Big).$$

5.4.2 Bivariate Case

The problem we tackle in this section is to formally verify that we know all the small integral roots $(x, y) \in \mathbb{Z} \times \mathbb{Z}$ of a pair of bivariate polynomials (P_1, P_2) on $\mathbb{Z}$. In other words, we consider the following kind of data:

$$\begin{cases} A \in \mathbb{N} \\ B \in \mathbb{N} \\ P_1 \in \mathbb{Z}[X, Y] \\ P_2 \in \mathbb{Z}[X, Y] \\ \mathcal{S} \subset \mathbb{Z} \times \mathbb{Z} \end{cases}$$

and we want to formally verify that we have

$$\forall (x, y) \in \mathbb{Z}^2, \quad (x, y) \in \mathcal{S} \iff \big(|x| \leqslant A \wedge |y| \leqslant B \wedge P_1(x, y) = 0 = P_2(x, y)\big). \tag{5.36}$$

For this purpose, we follow the same proof path as for univariate integral-roots certificates:

1. Define the type for bivariate integral-roots certificates as a `Record`;
2. Define the checker for these certificates as a Boolean function;
3. Prove the correctness of this checker (if it returns `true`, then (5.36) holds), using the uniqueness result for bivariate modular roots given by Lemma 5.5.

The formalization is very similar to the one we presented in Section 5.4 for the univariate case and the generalization to the bivariate case yields no technical difficulty, thanks to the material developed for proving the bivariate Lemma 5.5. Yet we will give all the main definitions and statements involved in the three steps of the formalization.

First, we consider the following type of certificate for the bivariate small-integral-roots problem:

```
Record bivCertif := BivCertif
{ bc_P1 : {bipoly Z}
; bc_P2 : {bipoly Z}
; bc_A : Z
; bc_B : Z
; bc_p : nat
; bc_k : nat
; bc_L : seq (Z * Z * bool)
}.
```

Then, we will say that one such certificate $(P_1, P_2, A, B, p, k, L)$ is valid if the following conditions are fulfilled:

$$p \text{ is a prime number,} \tag{5.37a}$$

$$\text{the integer } k \geqslant 0 \text{ satisfies } p^{2^k} > 2 \cdot A \text{ and } p^{2^k} > 2 \cdot B, \tag{5.37b}$$

and denoting $L_p = \{(u, v) \bmod p \,|\, \exists b \in \mathbb{B},\ (u, v, b) \in L\}$, we have:

$$\text{the elements of } L_p \text{ are pairwise distinct,} \tag{5.37c}$$

$$\forall (s,t) \in [\![0, p[\![^2,\ (s,t) \in L_p \iff P_1(s,t) \equiv 0 \equiv P_2(s,t) \pmod p, \tag{5.37d}$$

and for all $(u, v, b) \in L$, we have:

$$\det J_{P_1,P_2}(u,v) \not\equiv 0 \pmod p, \tag{5.37e}$$

$$|2 \cdot u| \leqslant p^{2^k} \ \wedge\ |2 \cdot v| \leqslant p^{2^k}, \tag{5.37f}$$

$$P_1(u,v) \equiv 0 \equiv P_2(u,v) \pmod{p^{2^k}}, \tag{5.37g}$$

$$b = \mathsf{true} \iff \big(|u| \leqslant A \ \wedge\ |v| \leqslant B \ \wedge\ P_1(u,v) = 0 = P_2(u,v)\big). \tag{5.37h}$$

These conditions are implemented in the form of a Boolean Coq function named "`biv_check : bivCertif -> bool`" which constitutes a *bivariate small-integral-roots certificates checker*.

The previously formulated Remarks 5.9, 5.10 and 5.11 are applicable in the context of the bivariate case, and the correctness lemma of the bivariate checker `biv_check` is as follows:

Lemma 5.8 (Correctness of the checker). *For all bivariate small-integral-roots certificate* `bc` $= (P_1, P_2, A, B, p, k, L)$ *such that* `(biv_check bc)` *holds, for all* $(z, w) \in \mathbb{Z}$ *we have the equivalence*

$$\big(|z| \leqslant A \wedge |w| \leqslant B \wedge P_1(z,w) = 0 = P_2(z,w)\big) \iff (z,w) \in \texttt{(bc\_roots bc)},$$

where `(bc_roots bc)` *is defined by "*`map1_filter2 (Z * Z) (bc_L bc)`*".*

Proof. The proof path is exactly the same as the one for Lemma 5.7, relying on the bivariate Lemma 5.5 in place of Lemma 5.2, and using twice the arithmetic Lemma 5.3, one for each variable. □

5.4.3 ISValP Certificates

As we have seen in Section 5.2.3, Coppersmith's technique reduces the ISValP problem to the bivariate small-integral-roots problem, and proceeds by implication, so that this technique can be used to ensure we have forgotten no solution of a given instance of ISValP.

Following the certificate-based approach (cf. Section 3.2.6), we implement a checker for ISValP certificates, built on top of our bivariate small-integral-roots certificates checker. We recall that this latter checker deals with certificates that can be generated using bivariate Hensel lifting, and that its correctness derives from the bivariate Hensel's lemma.

In the sequel, we will present the main steps we met for formalizing an ISValP certificates checker in Coq. We will thus present in Sections 5.4.3.1 and 5.4.3.2 two building blocks that are involved in this design, from the mathematical point of view to the Coq material itself, then Section 5.4.3.3 will be devoted to the presentation of the resulting formalization of the ISValP certificates, their checker and the corresponding correctness lemma.

5.4.3.1 Change of Polynomial Basis

From a formal point of view, a key step to encode this technique in a formal checker was to translate the fact the $v_1(X,Y)$ and $v_2(X,Y)$ are initially expressed in the polynomial basis $\{Q_{i,j}(X,Y)\}_{i,j}$, and compute their expression in the usual monomial basis. In particular, we could not have stored these polynomials in basis $\{X^iY^j\}_{i,j}$ directly, since the correctness of the method strongly relies on the presence of $M^{\alpha-i}$ inside the expression of $Q_{i,j}(X,Y)$. In other words, these powers of M have to be there, syntactically.

In our formalization, we perform the change of polynomial basis at stake in the following steps:

1. Given a bivariate polynomial v expressed with coefficients u_{ij} in basis $\{Q_{i,j}(X,Y)\}_{i,j}$ (i.e., assuming we have $v = \sum_{i\leqslant\alpha}\sum_j u_{ij}Q_{i,j}(X,Y)$), we first "lift" its coefficients u_{ij} to $u_{ij}M^{\alpha-i}$;
2. Then in the polynomial $\sum_{i\leqslant\alpha}\sum_j u_{ij}M^{\alpha-i}X^iY^j$ so obtained, we replace the indeterminate X with the polynomial Q to get
$$v = \sum_{i\leqslant\alpha}\sum_j u_{ij}M^{\alpha-i}Q^i(X,Y)Y^j;$$
this may be accomplished using a Horner-based polynomial composition.

We implemented the first step by defining a function `powers_alpha_pos` (in library `morepolyz`) for pre-computing lists of decreasing powers of M:

```
Definition Fpowers (M : Z) (x : Z * seq Z) : (Z * seq Z) :=
  let: (x1, x2) := x in (M * x1, x1 :: x2).
Definition powers_alpha_pos (alpha : positive) (M : Z) :=
  snd (iter_pos alpha (Z * seq Z)%type (Fpowers M) (1, [::])).
```

So we can get the list $(M^\alpha, M^{\alpha-1}, \ldots, M^1, 1)$ of size $\alpha+1$ by considering

`powers_alpha_pos (Psucc alpha) M`.

Then we define a recursive function `rec_precalc_alpha` that is in charge of multiplying this list component-wise with a bivariate polynomial `u` $\in \mathbb{Z}[X,Y]$ seen as a list of univariate polynomials in $\mathbb{Z}[Y]$:

```
Fixpoint rec_precalc_alpha
  (u : seq {poly Z}) (s : seq Z) {struct u} : seq {poly Z} :=
  match u, s with
    | u0 :: u', s0 :: s' => u0 * (s0%:P) :: rec_precalc_alpha u' s'
    | _, _ => [::]
  end.
```

where the notation `c%:P` denotes the constant polynomial with coefficient `c`.

We can combine both functions to define function `bipoly_precalc_alpha` (in library `morebipolyz`):

```
Definition bipoly_precalc_alpha
  (u : {bipoly Z}) (alpha : positive) (M : Z) :=
  let s := powers_alpha_pos (Psucc alpha) M in
  Poly (rec_precalc_alpha u s).
```

and we prove the corresponding correctness lemma:

```
Lemma bipoly_precalc_alpha_correct :
  forall (u : {bipoly Z}) (alpha : positive) (M : Z), M <> 0 ->
  let a := nat_of_pos alpha in
  bipoly_precalc_alpha u alpha M =
  \poly_(i < a.+1, j < size u`_i) (M ^+ (a - i) * u`_i`_j).
```

where the notation `M ^+ n` stands for the exponentiation (defined for any SSReflect `ringType`), `p`_i` denotes the `i`-th coefficient of list-based polynomial `p`, and `"\poly_(i < m, j < n) eij"` (with `i` and `j` being free variables in expression `eij`) is a shortcut we defined to denote the bivariate polynomial $\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} e_{ij} X^i Y^j$.

For the second step, we rely on the polynomial composition function `poly_comp` defined in SSReflect library `poly` that we briefly recall:

```
Section PolyCompose.
Variable R : ringType.
Implicit Types p q : {poly R}.
Definition poly_comp q p := (map_poly polyC p).[q].
End PolyCompose.
Notation "p \Po q" := (poly_comp q p) (at level 50).
```

This definition relies on a Horner evaluation of polynomial

`map_poly polyC p : {poly {poly R}}`

at `(q : {poly R})`, and we specialize it for `(R := {poly Z})`, namely for `p` and `q` having type `{bipoly R}`, so that it will involve an intermediate polynomial

`map_poly polyC p : {poly {poly {poly Z}}}`

To sum up, the material presented in this subsection will lead to local definitions

```
let v1 := (bipoly_precalc_alpha u1 alpha M) \Po Q in
let v2 := (bipoly_precalc_alpha u2 alpha M) \Po Q in
...
```

inside the reference implementation of our checker for ISValP certificates.

5.4.3.2 Computation of Weighted Norm 1

Then, a crucial step is to formalize the required material to be able to verify Equation (5.30) in an effective way. For this purpose, we consider the following definition of a "weighted norm-1" (called `bimaphorner` in the code), for a given $P \in \mathbb{Z}[X,Y]$ and $A, B \in \mathbb{N}$:

$$|P|(A,B) := \sum_i \sum_j |P_{ij}| A^i B^j \in \mathbb{N}.$$

Then we prove the following result:

Lemma 5.9 (`lez_bimaphorner`). *For any $P \in \mathbb{Z}[X,Y]$ and any $(A,B) \in \mathbb{Z}^2$, we have:*

$$\forall x, y \in \mathbb{Z}, \quad |x| \leqslant A \quad \wedge \quad |y| \leqslant B \quad \Longrightarrow \quad |P(x,y)| \leqslant |P|(A,B).$$

Thanks to this result (and considering $P := v_l$), we can notice that it is sufficient to have $|v_l|(A,B) < M^\alpha$ to make sure that Equation (5.30) holds.

And we derive the following extra result:

Lemma 5.10 (`modz_small_0`). *For any $N \in \mathbb{Z}$, we have:*

$$\forall z \in \mathbb{Z}, \quad |z| < |N| \quad \wedge \quad z \equiv 0 \pmod{N} \implies z = 0.$$

Finally we can combine Lemmas 5.9 and 5.10 to obtain the following key result:

Lemma 5.11 (`small_modular_roots_in_Z`). *For any $P \in \mathbb{Z}[X,Y]$ and any $(N,A,B) \in \mathbb{Z}^3$, we have:*

$$\begin{aligned} \forall x, y \in \mathbb{Z}, \quad |x| \leqslant A \quad \wedge \quad |y| \leqslant B \quad \wedge \quad |P|\,(A,B) < |N| \\ \wedge \quad P(x,y) \equiv 0 \pmod{N} \implies P(x,y) = 0. \end{aligned}$$

The definition `bimaphorner` presented in this subsection will be specialized and used for our reference (proof-oriented) implementation of our ISValP certificates checker, as shown by lines:

```
let Ma := Zpower_pos M alpha in
...
(bimaphorner Zabs A B v1 < Zabs Ma) &&
(bimaphorner Zabs A B v2 < Zabs Ma) &&
...
```

and Lemma 5.11 will be applied twice, one for each polynomial v_1 and v_2 obtained by the method presented in the previous Section 5.4.3.1.

5.4.3.3 ISValP Certificates Based on Hensel Lifting

First, we consider the following type of certificate for the ISValP problem presented in Section 5.2.3:

```
Record cert_ISValP : Type := Cert_ISValP
{ c_P : {poly Z} (* hence [Q(X,Y) := P(Y) - X] *)
; c_M : Z
; c_alpha : positive
; c_A : Z
; c_B : Z
; c_u1 : {bipoly Z} (* in basis [M^(alpha-i) * Q^i(X,Y) * Y^j] *)
; c_u2 : {bipoly Z} (* in basis [M^(alpha-i) * Q^i(X,Y) * Y^j] *)
; c_p : nat
; c_k : nat
; c_L : seq (Z * Z * bool)
}.
```

Next, using the material presented in Section 5.4.2, 5.4.3.1 and 5.4.3.2, we define the following certificates checker:

```
Definition check_ISValP (C : cert_ISValP) : bool :=
  let: Cert_ISValP P M alpha A B u1 u2 p k L := C in
  let Q := poly_cons P (bipolyC (-1)) in
  let v1 := (bipoly_precalc_alpha u1 alpha M) \Po Q in
  (* cf. Section 5.4.3.1 *)
  let v2 := (bipoly_precalc_alpha u2 alpha M) \Po Q in
  let Ma := Zpower_pos M alpha in
  let C' := BivCertif v1 v2 A B p k L in
  [&& 0 < M,
  bimaphorner Zabs A B v1 < Zabs Ma,
  (* cf. Section 5.4.3.2 *)
  bimaphorner Zabs A B v2 < Zabs Ma
  & biv_check C']. (* cf. Section 5.4.2 *)
```

where `(poly_cons P (bipolyC (-1)))` represents the polynomial "$P(Y) - X$". Finally, we derive the corresponding correctness lemma, whose statement is as follows:

```
Lemma check_ISValP_correct :
  forall C : cert_ISValP,
  check_ISValP C = true ->
  let: Cert_ISValP P M alpha A B u1 u2 p k L := C in
  forall n y : Z,
  let x := P.[y] - n * M in
  Zabs x <= A ->
  Zabs y <= B ->
  (x, y) \in map1_filter2 (Z * Z) L.
```

Remark 5.13 (Reasoning by implication). As we pointed it out in Remark 2.6, Coppersmith's technique proceeds by implication, so it finally leads to an inclusion in our correctness lemma `check_ISValP_correct`: all the solutions of the ISValP instance are gathered in the list L. Contrastingly, the converse

inclusion is mathematically trivial (it amounts to evaluating the polynomial P on each of the candidate roots), but implementing this ultimate verification in the ISValP checker would require the addition of Booleans inside the ISValP certificate, with another semantics than for those that are involved in the bivariate small-integral-roots certificates—we recall that the Booleans currently implemented in these latter certificates allows one to fully address the *equivalence* involved in (5.36).

5.4.4 A Modules-Based Formalization for Effective Checkers

The certificates checkers we have presented in Sections 5.4.1, 5.4.2 and 5.4.3 have been developed using some data structures that are suitable for the proofs, but not for effective computation. For instance, the implementation of polynomials in the SSReflect library `poly` relies on the constructor `Poly` (taking a list of coefficients as an argument), which is not computational. We thus need to re-implement our certificates checkers with effective data structures.

In particular, we want to define generic checkers that can be instantiated with the desired integer operations and implementation of polynomials, while being able to derive their correctness by directly relying on the reference implementation based on SSReflect datatypes. We chose to rely on the module system of Coq to achieve this genericity.

5.4.4.1 Presentation of the Main Abstract Interfaces

To begin with, we define abstract interfaces (`Module Type`s) for computational rings, rings of univariate polynomials, rings of bivariate polynomials, as well as a specific interface for the ring of integers, which will handle some specific functions such as modulo and absolute value in addition to the ring operations. For instance, here is an excerpt of the Coq code defining the first interface in the hierarchy:

```
Module Type CalcRingSig.
Parameter T : Type.
Parameter R : comRingType.
Parameter toR : T -> R.
Parameter ofR : R -> T.
Parameter ofR_inj : forall r : R, toR (ofR r) = r.
Parameter t0 : T.
Parameter t1 : T.
Parameter topp : T -> T.
Parameter tadd : T -> T -> T.
(* tsub, tmul, tnatmul, tnatexp omitted *)
Parameter toR_0 : toR t0 = 0%R.
Parameter toR_1 : toR t1 = 1%R.
Parameter toR_opp : forall x : T, toR (topp x) = (- toR x)%R.
Parameter toR_add :
  forall x y : T, toR (tadd x y) = (toR x + toR y)%R.
(* more omitted *)
End CalcRingSig.
```

The role of this interface is to gather an implementation of "ring operations" on a given type `T`, which are proved correct with respect to the operations on a specific `R` of type `comRingType`, the SSReflect structure for commutative rings. Note that we consider two functions "`toR : T -> R`" and "`ofR : R -> T`" from one type to the other, assuming that `ofR` is injective, unlike `toR` in general. The fact we do not assume bijectivity is central to be able to consider datatypes that provide multiple representations for a given single mathematical object (e.g., in `bigZ` integers, zero admits at least two representations).

Then we define an interface for univariate polynomials `CalcPolySig` on a computational ring `CalcRingSig`:

```
Module Type CalcPolySig (A : CalcRingSig) <: CalcRingSig.
Include CalcRingSig.
Parameter tpolyC : A.T -> T.
Parameter rpolyC : A.R -> R.
Parameter tpolyX : T.
Parameter rpolyX : R.
Parameter tpolycons : A.T -> T -> T.
Parameter rpolycons : A.R -> R -> R.
Parameter tderiv1 : T -> T.
Parameter rderiv1 : R -> R.
Parameter thorner : T -> A.T -> A.T.
Parameter rhorner : R -> A.R -> A.R.
Parameter toR_polyC : forall c, toR (tpolyC c) = rpolyC (A.toR c).
Parameter toR_polyX : toR tpolyX = rpolyX.
Parameter toR_polycons :
  forall x t, toR (tpolycons x t) = rpolycons (A.toR x) (toR t).
Parameter toR_deriv1 : forall t, toR (tderiv1 t) = rderiv1 (toR t).
Parameter toR_horner :
  forall t x, A.toR (thorner t x) = rhorner (toR t) (A.toR x).
End CalcPolySig.
```

First, the `Include` invocation above enforces the fact that any implementation of interface `CalcPolySig` must implement all the parameters from `CalcRingSig`. This amounts to saying that the polynomials on a ring themselves constitute a ring. Then, the interface `CalcPolySig` specifies four additional "constructors" (for constant polynomials, indeterminate, concatenation, derivative) and Horner evaluation, each with two versions (computational and proof-oriented).

Then we can specify bivariate polynomials as an interface called `CalcBiPolySig`, whose definition starts by iterating the `CalcPolySig` functor twice on the module argument `A`:

```
Module Type CalcBiPolySig (A : CalcRingSig).
Declare Module Univ : CalcPolySig A.
Include CalcPolySig Univ.
Parameter tpolyY : T.
Parameter tderiv2 : T -> T.
Parameter rderiv2 : R -> R.
(* functions for Horner evaluation omitted *)
Parameter tbimaphorner : (A.T -> A.T) -> A.T -> A.T -> T -> A.T.
Parameter rbimaphorner : (A.R -> A.R) -> A.R -> A.R -> R -> A.R.
```

```
Parameter tliftalpha : T -> positive -> A.T -> T.
Parameter rliftalpha : R -> positive -> A.R -> R.
(* properties omitted *)
End CalcBiPolySig.
```

Finally, we define the interface `RingIntSig` for the ring of integers by extending the interface `CalcRingSig` with functions such as exponentiation, modulo, absolute value, and order predicates, then we specialize the parameter `(R := Z)` and the operations on `R`, so that the "modular proofs" that are developed upon `CalcRingIntSig` can rely on the "reference proofs" presented in Sections 5.3, 5.4.1, 5.4.2 and 5.4.3, which are based on the type `Z` and the related operations from `ZArith`:

```
Module Type CalcRingIntSig :=
  RingIntSig with Definition R := Z_comRingType
  with Definition rexp := Zpower_pos
  with Definition req := @eq_op Z_eqType
  with Definition rlt := Zlt_bool
  with Definition rle := Zle_bool
  with Definition rabs := Zabs
  with Definition rmod := Zmod
  with Definition nat2R := Z_of_nat.
```

We provide two[4] *implementations* of this interface `CalcRingIntSig`, through modules `CalcRingZ` and `CalcRingBigZ`. (To sum up, module `CalcRingZ` straightforwardly implements the same integer operations as those we have chosen for the proofs, while module `CalcRingBigZ` relies on the `BigZ` library for efficient multiple-precision integer arithmetic, based on machine integers.)

5.4.4.2 Implementation of Generic Effective Checkers

This subsection is devoted to our modular implementation of effective checkers for small univariate (resp. bivariate) integral roots certificates that are gathered in modules `CalcUnivHensel` and `CalcBivHensel`, as well as an effective checker for ISValP certificates in module `CalcISValP`.

These checkers consist of a computational version of the checkers presented in Sections 5.4.1, 5.4.2 and 5.4.3.3, and their formalization relies on the modular hierarchy mentioned in previous Section 5.4.4.1.

In each of the modules `CalcUnivHensel`, `CalcBivHensel` and `CalcISValP` that have a similar structure, the soundness of the checkers is expressed in the form of two results:

```
Lemma tCheck_correct : forall t : tC, tCheck t = rCheck (toC t).
Lemma rCheck_correct : forall c : rC, rCheck c = true -> rValid c.
```

First, `tCheck_correct` ensures each computational checker returns `true` if and only if the corresponding non-computational checker returns `true`. Second, `rCheck_correct` ensures if the considered (non-computational) checker returns `true` for a given certificate `c`, then this certificate relevantly describes all the solutions of the problem at stake.

[4] A third one will be presented in Section 5.5.3.

In the rest of this section, we will give more details on the various steps we met during this modular formalization, focusing especially on the bivariate case.

So we start by defining a parameterized module called

```
Module CalcBivHensel (A : CalcRingIntSig).
```

First, we need to have type "`seq (A.T * A.T * bool)`" and some related effective functions for the uniqueness and belonging predicates. We provide this material in a generic way by defining a few modules beforehand:

```
Module CalcProd (A B : CalcEqSig) <: CalcEqSig.
Definition T := (A.T * B.T)%type.
Definition teq x y := A.teq x.1 y.1 && B.teq x.2 y.2.
Definition Q := [eqType of (A.Q * B.Q)%type].
Definition toQ : T -> Q := fun x => (A.toQ x.1, B.toQ x.2).
Definition ofQ : Q -> T := fun x => (A.ofQ x.1, B.ofQ x.2).
Lemma ofQ_inj : cancel ofQ toQ.
Lemma toQ_eq : forall m n, teq m n = ((toQ m) == (toQ n)).
End CalcProd.
```

which strongly relies on the formalization of types with decidable equality in the SSReflect library `eqtype`. Then:

```
Module CalcList (A : CalcEqSig) <: CalcEqSig.
Definition T := seq A.T.
Fixpoint in_seq (x : A.T) (s : seq A.T) {struct s} : bool.
Fixpoint notin_seq (x : A.T) (s : seq A.T) {struct s} : bool.
Fixpoint uniq_seq (s : seq A.T) {struct s} : bool.
Section Defix.
  Variables (U : Type) (EQ : U -> U -> bool).
  Fixpoint eq_seq s1 s2 {struct s2} : bool.
End Defix.
Definition teq : T -> T -> bool := eq_seq A.teq.
Definition Q := [eqType of seq A.Q].
Definition toQ : T -> Q := map A.toQ.
Definition ofQ : Q -> T := map A.ofQ.
Lemma ofQ_inj : cancel ofQ toQ.
Definition Tb := seq (A.T * bool).
Definition Qb := [eqType of seq (A.Q * bool)].
Definition toQb : Tb -> Qb := map (fun e => (A.toQ e.1, e.2)).
Definition ofQb : Qb -> Tb := map (fun e => (A.ofQ e.1, e.2)).
Lemma ofQb_inj : cancel ofQb toQb.
Lemma toQ_eq : forall x y, teq x y = eqseq (toQ x) (toQ y).
Lemma spec_in : forall x s, in_seq x s = ((A.toQ x) \in (toQ s)).
Lemma spec_notin :
  forall x s, notin_seq x s = ((A.toQ x) \notin (toQ s)).
Lemma spec_uniq : forall s, uniq_seq s = uniq (toQ s).
Definition AtoQb := (fun e : A.T * bool => (A.toQ e.1, e.2)).
(* more lemmas omitted *)
End CalcList.
```

which provides support for the Boolean predicates on lists that are required in our modular formalization. Now in module `CalcBivHensel`, we can process

```
Module ZYX := CalcBiPolyFull A.
Module ZZ := CalcProd A A.
Module LZZ := CalcList ZZ.
Module LZ := CalcList A.
```

Note that the module `LZ` is useful mostly for proving purposes. Then we can define the following types and functions:

```
Record tC := Build_tC
{ c_P1 : ZYX.T
; c_P2 : ZYX.T
; c_B1 : A.T
; c_B2 : A.T
; c_p : nat
; c_k : nat
; c_L : LZZ.Tb
}.
```

which correspond to the inductive type of *bivariate small-integral-roots certificate*, while the effective checker for such certificates has type

```
Definition tCheck : tC -> bool. (* definition omitted, cf. (5.37) *)
```

Then the following addresses the correctness of this checker:

```
Definition rC := bivCertif.
Definition rCheck : rC -> bool := biv_check.
Definition rP1 : rC -> {bipoly A.R} := bc_P.
Definition rP2 : rC -> {bipoly A.R} := bc_Q.
Definition rB1 : rC -> A.R := bc_A.
Definition rB2 : rC -> A.R := bc_B.
Definition rRoots : rC -> LZZ.Q := bc_roots.
Definition rBounded : A.R -> A.R -> bool :=
  fun a z => in (Zabs z <= Zabs a).
Definition toC : tC -> rC. (* definition omitted *)
Definition rValid (c : rC) : Prop :=
  forall z w : A.R,
  [&& rBounded (rB1 c) z, rBounded (rB2 c) w
    & biv_root (rP1 c) (rP2 c) z w]
  <-> (z,t) \in (rRoots c).
Lemma tCheck_correct : forall t : tC, tCheck t = rCheck (toC t).
Lemma rCheck_correct : forall c : rC, rCheck c = true -> rValid c.
End CalcBivHensel.
```

Note that the functions with a `t` prefix are related to the computational type `T` while those with a `r` prefix are "proof-oriented". To be more precise, the effective checker "`tCheck : tC -> bool`" on the certificate type `tC` is proved correct with respect to the reference checker "`rCheck : rC -> bool`," whose correctness claim involves the logical predicate of validity "`rValid : rC -> bool`".

The lemma `rCheck_correct` is directly proved using the mechanized proof of Lemma 5.8 called `small_roots_when_bc_is_valid` in the Coq sources, while for `tCheck_correct`, we heavily rely on the correctness lemmas that we added in our hierarchy, leading to dozens of lines of rewriting tactics.

Likewise, we derive a modular definition of effective certificate checkers for the univariate small-integral-roots problem (in module `CalcUnivHensel`), as well as for the ISValP problem (in module `CalcISValP`).

5.4.5 Some Concrete Examples of Use

We now present the characteristics of four ISValP instances that cover the entire spectrum of situations that may occur for our targeted application (i.e., solving the TMD using SLZ).

For each of these instances, we successively give in Table 5.1 the function at stake, the target precision prec as well as the parameter prec′ that corresponds to the variable q in Definition 2.17, the expansion point x_0, the (normalized) approximation polynomial $P \in \mathbb{Z}[X]$, the modulo M, and the bounds A and B.

These instances have been chosen so that

- #1, #2 and #3 deal with the binary64 format, while #4 deals with the binary128 format;
- #1 contains a hardest-to-round-point, unlike others instances;
- #3 and #4 deal with the Approximate-TMD, with a value chosen for prec′ much larger than $2 \times$ prec.

Problem ID	f	prec	prec′	x_0	$\deg(P)$	$\max_i(\lvert P_i \rvert)$	M	A	B
#1	exp	53	100	$54582361821388000/2^{53}$	2	$\lessapprox 1.68\times 2^{237}$	2^{185}	2^{139}	2^{12}
#2	exp	53	100	-2^{-1}	2	$\lessapprox 1.22\times 2^{237}$	2^{185}	2^{139}	2^{12}
#3	exp	53	300	$1+2^{-21}$	12	$\lessapprox 1.36\times 2^{996}$	2^{942}	2^{696}	2^{32}
#4	exp	113	3000	$1+2^{-21}$	90	$\lessapprox 1.36\times 2^{13661}$	2^{13547}	2^{10661}	2^{72}

Table 5.1 – *Short description of four instances of ISValP*

Then, for a given parameter α, SLZ is in charge of producing polynomials v_1 and v_2, from which Hensel lifting is executed, which leads to complete ISValP certificates as described in Table 5.2. The columns $\lvert v1 \rvert\,(A,B)$ and $\lvert v_2 \rvert\,(A,B)$ give the order of magnitude of the weighted norm-1 of both v_1 and v_2, following the definition we have given in Section 5.4.3.2: these values are large but indeed less than M^α, which is verified by the ISValP checker. Then, the subsequent columns describe the parameters related to Hensel lifting: the prime p, the integer k and the list L (whose size is denoted by $\#\,L$). We finally give the timings for the certificate to be accepted by `Coq`: these benchmarks have been computed on the same machine as for Section 4.4: a 8-core computer, Intel(R) Xeon(R) CPU E5520 @ 2.27GHz with 16 GB of memory, using Coq 8.3pl4, `vm_compute` and the implementation of `check_ISValP` that takes into account the optimizations that we will present in Section 5.5.2.

We can notice the following points:

- For the ISValP instance #2, changing the prime p has an impact on the size of L (as stated in Remark 5.12), and thereby on the parsing timing of the certificate. However, the timing that is necessary for the ISValP checker to accept the certificate appears to be decorrelated with the size of L, while this timing slightly increases when p increases;

Pb ID	α	$\lvert v_1\rvert(A,B)$	$\lvert v_2\rvert(A,B)$	M^α	p	k	$\#L$	time to parse	time to return true
#1	2	$\lessapprox 1.23\times 2^{354}$	$\lessapprox 1.77\times 2^{353}$	2^{370}	5	6	1	0.096s	0.092s
#2	2	$\lessapprox 1.02\times 2^{356}$	$\lessapprox 1.05\times 2^{356}$	2^{370}	7	6	2	0.132s	0.112s
					3	7	1	0.112s	0.092s
					23	5	0	0.088s	0.172s
#3	4	$\lessapprox 1.84\times 2^{3766}$	$\lessapprox 1.13\times 2^{3767}$	2^{3768}	5	9	0	0.420s	2.348s
#4	6	$\lessapprox 1.68\times 2^{81197}$	$\lessapprox 1.81\times 2^{81200}$	2^{81282}	5	14	0	17.4s	3h12m42s

Table 5.2 – *Short description of some ISValP certificates with their verification timing*

- For the ISValP instance #3, we can extrapolate from the obtained timings to estimate the time for verifying 2^{30} ISValP certificates, in order to address the full range of an exponent in binary64: $2.768\,\mathrm{s} \times 2^{20} \approx 34\,\mathrm{days}$, which is feasible;
- Finally the verification of the fourth considered instance of ISValP is expensive, and we expect this is due to our naive implementation of polynomial multiplication, on which depends our implementation of bivariate polynomial composition. We have indeed performed some experiments related to this ISValP instance that show that the two compositions involved in the verification take over 95% of the timing. So it should be possible to get an appreciable speed-up by implementing optimized operations on polynomials, for instance using a Karatsuba-based approach [95]. This may also need to change the data structures chosen for our computational bivariate polynomials.

5.5 Technical Issues

In this section we discuss the relevancy of our formalization choices on which relies the material that we have presented in Sections 5.3 and 5.4.

5.5.1 Formalization Choices

First of all, we focus on some key choices we have made for our formalization. In particular, for each required core concept, we will mention the main possibles choices we have considered and discuss some of their advantages and drawbacks.

Relative Numbers

As regards the ring $\mathbb{Z}$ of relative numbers, at least two different formalizations are currently available:

- The library `ZArith` on binary integers from the `Coq` standard library;
- Cyril Cohen's `ssrint` library[5] (formerly `zint`).

[5] https://gforge.inria.fr/scm/viewvc.php/trunk/Saclay/Ssreflect/theories/ssrint.v?view=markup&revision=2029&root=coq-contribs

For the sake of completeness, we briefly recall each definition:

In library `ZArith`, the type `Z` is defined as an inductive type for signed, binary integers:

```
Inductive Z : Set :=
  | Z0 : Z
  | Zpos : positive -> Z
  | Zneg : positive -> Z.
```

```
Inductive positive : Set :=
  | xI : positive -> positive
  | xO : positive -> positive
  | xH : positive.
```

(For example, the Coq term "`Zneg (xO (xO (xI xH)))`" will represent the negative number -12, whose absolute value is 1100 in radix 2.)

On the other hand, the library `int` relies on the following definitions:

```
Inductive int : Set :=
  | Posz : nat -> int
  | Negz : nat -> int.
Coercion Posz : nat >-> int.
```

While the formalism of `int` may be quite handy to use, notably thanks to the presence of the `Coercion` above, this library does not provide yet any material on modular arithmetic.

We thus chose to use `ZArith` for our formalization, where a modulo operation "`Zmod : Z -> Z -> Z`" is available with a comprehensive set of related lemmas. All the new results that make `ZArith` fully compatible with SSReflect are gathered in our theory `ssrzarith`.

Note that `ZArith` actually provides two implementations for the modulo: `ZOmod`, very similar to OCaml's modulo, and `Zmod`, whose values are always nonnegative for a positive second argument, as shown by lemma

```
Z_mod_lt : forall a b : Z, (b > 0) -> (0 <= a mod b < b)
```

We will take advantage of this convention when deriving the lemma that links `Zmod` and SSReflect's `modn` on natural numbers. Note however that in the case where the second argument is zero (and the first one is nonzero), these two functions will differ:

```
Eval compute in (Zmod 5 0)%Z.
```

returns "`0 : Z`", while

```
Eval compute in (modn 5 0)%N.
```

returns "`5 : nat`". Nevertheless, this is not distracting for our formalization since we just need modular reduction modulo $q = p^{2^k} \geqslant p \geqslant 2 > 0$, for which both functions coincide, as stated by the Coq lemma `Z_of_nat_moduli` that was presented in Section 5.3.1.

Finally, while an alternative approach for dealing with modular arithmetic could be working *directly* in the quotient rings $\mathbb{Z}/q\mathbb{Z}$, we cannot rely on this single approach throughout the formalization. Indeed, the problems we are focusing on typically rely on *integral roots* (in $\mathbb{Z}$), for polynomials that are initially expressed with *integer coefficients*. However, as we pointed it out in Section 5.3.1, our proofs require the availability of the *rings* $\mathbb{Z}/q\mathbb{Z}$, provided in the SSReflect library `zmodp`.

Natural Numbers

In our different certificates, we chose to consider natural numbers `p` and `k` as inhabitants of type `nat` (Peano integers) while other choices were possible, notably the type `N` for nonnegative integers based on the type `positive`, which comes with somewhat efficient operations. This choice was motivated by the three following reasons:

- Considering `(p : nat)` allows one to use the Boolean primality predicate provided in the SSReflect libraries, which is fully computational and has type

  ```
  prime : nat -> bool.
  ```

- Considering `(k : nat)` contributed to ease the formalization of the iterated Hensel lifting algorithms and their correctness proofs by induction.

- The integers `p` and `k` that are to be considered for solving instances of the ISValP problem are very small (i.e., with an order of magnitude much lower than 1000), as shown by our four examples of ISValP certificates presented in Section 5.4.5.

However, as far as the parameter $\alpha \in \mathbb{N}^*$ is considered, we chose the type `positive` instead, since "most often, α growing to infinity is asymptotically the optimal choice" for Coppersmith's technique [157].

Bivariate Polynomials

Formalizing bivariate Hensel lifting requires to manipulate bivariate polynomials on the ring $\mathbb{Z}$. We chose to formalize bivariate polynomials on an arbitrary SSReflect ring `(R : ringType)` as polynomials with polynomial coefficients on `R`, that is, `{bipoly R}` stands for `{poly {poly R}}`.

This iterated definition is convenient to inherit some ring operations and support theorems from the SSReflect library `poly`. In particular, the outermost `{poly _}` type immediately comes with a notation `'X` which has type `{bipoly R}`, while the monomial `'Y` can easily be defined by injecting the innermost `('X : {poly R})` in `{bipoly R}` as a constant (bivariate) polynomial. In other words, this straightforward definition for `{bipoly R}` amounts to posing $R[X,Y] = (R[Y])[X]$, which is coherent with the usual indexed writing

$$P(X,Y) = \sum_i \sum_j \left(P_{ij} X^i Y^j\right),$$

where the summation on the first index i related to powers of X is written first (i.e., outermost).

Note also that the intrinsic asymmetry of this definition for `{bipoly R}` combined with the representation of univariate polynomials provided by the SSReflect type `{poly _}`, namely finite lists of coefficients, can be viewed as a compromise between dense and sparse bivariate polynomial representations.

The Indeterminates

During the formalization, we have been led to inverse the order of indeterminates with respect to [160, Section 3.2] that was considering $Q_{i,j}(X,Y) = X^iQ^j(X,Y)M^{\alpha-j}$. Indeed, the configuration with $Q(X,Y) = P(Y) - X$ and $Q_{i,j}(X,Y) = Q^i(X,Y)M^{\alpha-i}Y^j$ appeared to be necessary given the context of the formalization, and at the same time this rewording also is more convenient for defining Q.

To be more precise, we mentioned in Section 5.4.3.1 the use of polynomial composition to substitute $Q(X,Y)$ for one indeterminate. But `poly_comp` (the SSReflect definition for polynomial composition), in the context of bivariate polynomials presented in Section 5.5.1, is applicable only for replacing the indeterminate X.

Moreover, the bivariate polynomial $Q(X,Y) := P(Y) - X$ can be straightforwardly defined and computed using the "constructors" for polynomials (e.g., `poly_cons P (bipolyC (-1))`) rather than computing $P(X) - Y$ using a `map`-based definition to turn $P \in \mathbb{Z}[X]$ into $P(X) \in \mathbb{Z}[X,Y]$.

Order-2 Square Matrices

As mentioned in Section 5.3.3 we rely on order-2 square matrices in the presentation of the bivariate proof. Although not essential to derive the proof, this allows for more concise expressions and proof steps. We thus developed some Coq material specific to 2-by-2 matrices, including a version of Cramer rule with moduli, whose correctness lemma may be summarized as follows:

$$\begin{aligned}&\forall p \in \mathbb{P}, \quad \forall k \in \mathbb{N}, \quad \forall A \in \mathcal{M}_2(\mathbb{Z}), \quad \forall u \in \mathbb{Z}^2 = \mathcal{M}_{2,1}(\mathbb{Z}),\\ &\det(A) \not\equiv 0 \pmod{p} \implies A \times \left(A^{-1}_{p^{2^{k+1}}} \times u\right) \equiv u \pmod{p^{2^{k+1}}}. \end{aligned} \tag{5.38}$$

Note that we did not use the generic SSReflect library `matrix`, given that the proof path we followed in Section 5.3.3, focusing on bivariate polynomials, does not involve indexed general terms. Moreover, the kind of definition we chose for these 2-by-2 matrices (viz., a polymorphic `Record` with 4 fields), makes it possible to get a computational version of these matrices for free. The availability of these 2-by-2 computational matrices will be shown useful at Section 5.5.2.

Choice of a Modularization Mechanism

Finally, for the same reasons mentioned in Section 4.3.1, we chose to rely on the Coq module system for developing the material presented in Section 5.4.4.

5.5.2 Optimizations

In this section, we briefly summarize the optimizations that we implemented to increase the efficiency of our effective certificates checkers. Note that these optimizations lead to changes only for the implementation of the computational checkers, so the reference, proof-oriented checkers are unchanged.

Horner Evaluation of the Jacobian Determinant

To begin with, we describe two simple optimizations related to the verification of condition (5.37e) (ensuring hypothesis (5.19) is fulfilled).

First, the quantity $\det J_{P_1,P_2}(u,v)$ can be computed in several ways: we can either compute the determinant $\Delta(X,Y) = \frac{\partial P_1}{\partial X} \times \frac{\partial P_2}{\partial Y} - \frac{\partial P_2}{\partial X} \times \frac{\partial P_1}{\partial Y}$ and deduce the quantity $\Delta(a,b)$, or perform the Horner evaluation beforehand on each of the derivatives and compute the determinant of the matrix so obtained. We first implemented the former calculation, while the latter one has the advantage of requiring no multiplication of polynomials, since the determinant acts on a matrix in $\mathcal{M}_2(\mathbb{Z})$ instead of $\mathcal{M}_2(\mathbb{Z}[X,Y])$. For implementing this optimization, we benefit from our formalization of 2-by-2 matrices with coefficients in `(T : Type)`, which is applicable for the modular hierarchy we presented in Section 5.4.4. To sum up, using the ambient module `ZYX` that is an instance of `CalcBiPolySig`, we add a let-in for pre-computing `JM := ZYX.tmatJ P1 P2` (i.e., J_{P_1,P_2}), then we compute "`A.tdet (ZYX.tmateval JM u v)`" (i.e., $\det\left(J_{P_1,P_2}(u,v)\right)$) for each (u,v,b) stored in the list L.

Second, the Jacobian determinant is useful for checking whether all the elements of list L are single roots modulo p, which will always be the case if there are none. This is illustrated by the following result from the SSReflect library `seq`:

```
Lemma all_nil :
  forall (T : Type) (a : pred T), all a [::] = true.
```

(where the notation `[::]` stands for the empty list). As a result, we can add an if-then-else construct that tests if L is the empty list, and move the let-in that computes J_{P_1,P_2} inside the appropriate branch of the if-then-else. Consequently when using call-by-value reduction, especially through the Coq tactics `compute` and `vm_compute`, we obtain a speed-up in cases where the list L of the certificate is empty, given that the Jacobian matrix is no more evaluated.

Performing a Modular Reduction Beforehand

Moreover, the Coq function `calcBivCheck_modp` whose purpose is to check whether the conditions (5.37c) and (5.37d) related to the list

$$L_p = \{(u,v) \bmod p \,|\, \exists b \in \mathbb{B},\ (u,v,b) \in L\}\,,$$

are fulfilled, involves the computation of Boolean tests of kind

```
let zp := (A.nat2T p) in
  eqb (LZZ.in_seq (s,t) Lp)
    (A.teq (A.tmod P1.2[s,t] zp) A.t0 &&
     A.teq (A.tmod P2.2[s,t] zp) A.t0)
```

that is to say, $(s,t) \in L_p \overset{?}{\iff} P_1(s,t) \equiv 0 \equiv P_2(s,t) \pmod p$. As in practice, the coefficients of P_1 and P_2 will be somewhat big compared with p, we can optimize these calculations by performing the modular reduction "`A.tmod _ zp`" beforehand, that is, applying the modulo operation on each individual coefficients of P_1 and P_2, and unroll the polynomial evaluation on (s,t) afterwards.

5.5.3 IPPE, An Implementation of Integers with Positive Exponent

We notice that the coefficients of the approximation polynomials that will be typically involved in the final verification chain for SLZ will be floating-point numbers in radix-2, and that these polynomials will be scaled by a large power of 2 for generating an instance of ISValP (where the polynomial P has integer coefficients). For instance, in the example of the fourth certificate presented in Table 5.1, the power of 2 to be considered is 2^{13660}, leading to a polynomial on $\mathbb{Z}$ whose largest coefficient is $M \times 2^{10629}$ where M is an odd integer in $[\![2^{3032}, 2^{3033}[\![$.

Consequently, we decided to write another implementation of the integer coefficients for our ISValP checker: instead of using mere `bigZ` numbers, we focus on pairs $(m, e) \in$ `bigZ` $\times$ `bigN` corresponding to unevaluated floating-point numbers $m \times 2^e$ with $e \geqslant 0$. We implement this with a special focus on genericity. First, we define an abstract interface

```
Module Type CalcRingExpo (Import C : FloatCarrier).
```

that gathers several definitions and properties linked to "`typeN : Type`," which is to be instantiated with a type of efficient natural numbers for the exponent, e.g., `bigN`. As regards the parameter `C`, its signature is based on the `FloatCarrier` abstract interface provided in the CoqInterval library [118]. Then, we define the main module

```
Module CalcRingIPPE
  (Import C : FloatCarrier)
  (Import E : CalcRingExpo C)
    <: CalcRingIntSig.
```

that depends on both `FloatCarrier` (which notably specifies the radix) and `CalcRingExpo` interfaces, and which implements all the required operations with respect to the `CalcRingIntSig` interface. In particular, the type `T` of "*integers plus positive exponent*" can be defined as follows:

```
Notation typeZ := smantissa_type.
Record T := TZN { TZ : typeZ; TN : typeN }.
```

where the type `smantissa_type` is a member of the `FloatCarrier` signature. We implement the evaluation of these abstract integers in the form of a function named "`toTZ : T -> typeZ`," which can afterwards be composed by the function "`MtoZ : typeZ -> Z`" provided in the `FloatCarrier` interface, when defining:

```
Definition toR : T -> R := fun x => MtoZ (toTZ x).
```

We then define `tadd`, `tmul` and `texp` as (exact) floating-point addition, multiplication and exponentiation on the type `T`, as well as opposite `topp`, subtraction `tsub`, absolute value `tabs`, modular remainder `tmod`, and Boolean comparison predicates `teq`, `tlt` and `tle`. Then we prove the correctness of all these functions with respect to the the reference integers `(R := Z)` by means of "*homomorphism lemmas,*" expressed in terms of `toR`.

This optimized implementation led to an appreciable twofold speedup for the certificate #4 presented in Table 5.2, which is coherent with the kind of simplification that is induced in the computations.

Finally, Figure 5.1 summarizes the dependencies between the different theories belonging to the CoqHensel library described in this chapter.

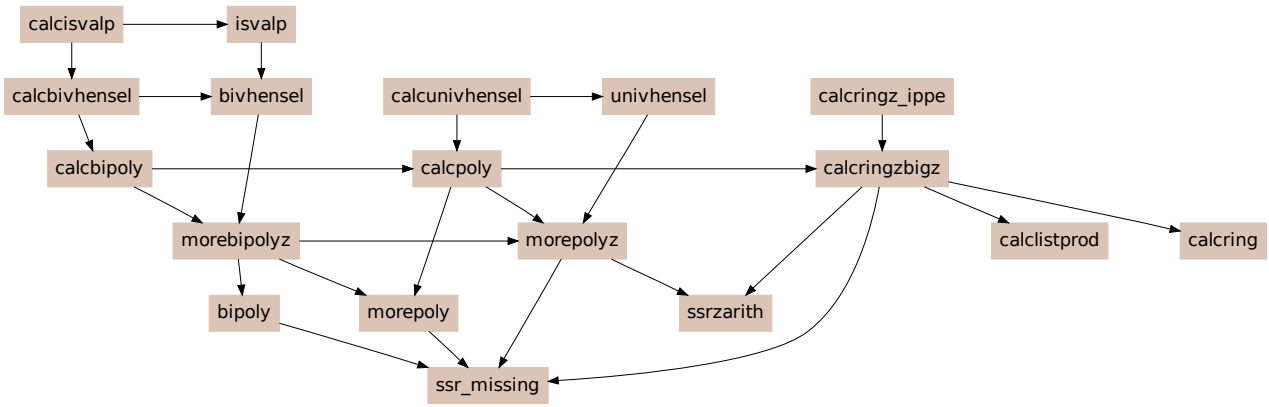

Figure 5.1 – *Dependency graph of the CoqHensel theories*

5.6 Conclusion and Perspectives

We have implemented, formally verified and made effective three interrelated certificate checkers in Coq. In particular, our ISValP certificate checker is suitable to check the claimed solutions to the *Integer Small-Value Problem*, typically obtained by a computational-expensive algorithm based on Coppersmith's technique, such as SLZ. In the context of the TMD problem (see Section 2.4), this allows one to ensure that no bad case for correct rounding has been forgotten. In particular, the complete proof of such a statement will result from the combination of the CoqHensel machinery presented here with the one that is under development in the CoqApprox library, in order to fully formally certify the whole SLZ algorithmic chain for solving the TMD. We notably expect that the formalization of "*Integers Plus Positive Exponent*" that we have developed for the sake of efficiency of the coefficient operations will ease the link with CoqApprox, since both libraries are built upon the CoqInterval formalism.

Both from the point of view of the implementation and the formal proof of correctness, we recall that we have formalized our ISValP checker upon our *bivariate small-integral-roots certificates checker*, the latter taking advantage of the semantics of Hensel lifting. This formalization requires several notions, such as Taylor theorem for bivariate polynomials, the Jacobian matrix of a pair of bivariate polynomials, Cramer's rule for order-2 matrices in modular arithmetic, and the weighted-norm-1 of a bivariate polynomial, which have been implemented in Coq.

As regards the univariate case, we have also formalized the notion of valuation for a polynomial on an arbitrary ring and formally proved Lemma 5.1. As mentioned in Remark 5.1, this allows one to use our univariate small-integral-roots certificates checker to address the univariate whole-integral-roots problem,

which is an interesting application in itself. Furthermore, we would like to investigate the possible extension of this formalization to address the problem of rational roots of polynomials.

As detailed in previous Section 5.5, a key part of the work consists of making a number of formal development choices, taking into account both proof and computation aspects. In particular, we wanted to obtain a generic prototype that allows one to easily swap optimized implementations of the various required building blocks: we achieve this genericity by means of Coq parameterized modules.

A minor point is the time to check some ISValP certificates (3 hours). As mentioned in previous Section 5.4.5, the current order of magnitude of this computation timing is certainly due to our naive implementation of polynomial multiplication. As a short-term future work, we thus plan to implement the Karatsuba algorithm [95] for polynomials in order to speedup the computation for such "extreme" certificates, while keeping the confidence we have in our formally verified ISValP certificates checker.

Chapter 6

Augmented-Precision Algorithms for Correctly-Rounded 2D Norms

This chapter is based on a joint work that resulted in a publication in the proceedings of the IEEE ARITH 2011 conference [29].

We define an "augmented precision" algorithm as an algorithm that returns, in precision-p floating-point arithmetic, its result as the unevaluated sum of two floating-point numbers, with a relative error of the order of 2^{-2p}. Assuming an FMA instruction is available, we perform a tight error analysis of an augmented precision algorithm for the square root, and introduce two slightly different augmented precision algorithms for the 2D norm $\sqrt{x^2+y^2}$. Then we give tight lower bounds on the minimum distance (in ulps) between $\sqrt{x^2+y^2}$ and a midpoint when $\sqrt{x^2+y^2}$ is not itself a midpoint. This allows us to determine cases when our algorithms make it possible to return correctly-rounded 2D norms.

6.1 Introduction

In some applications, just returning a floating-point approximation y_h to the exact result y of a function or arithmetic operation may not suffice. It may be useful to also return an estimate y_ℓ of the error (i.e., $y \approx y_h + y_\ell$). For simple enough functions (e.g., addition or multiplication), it is even possible to have $y = y_h + y_\ell$ exactly. Having an estimate of the error makes it possible to re-use it later on in a numerical algorithm, in order to at least partially compensate for that error. Such *compensated* algorithms have been suggested in the literature for summation of many floating-point numbers [91, 4, 144, 145, 141, 147], computation of dot products [138], and evaluation of polynomials [66].

We will call *augmented-precision algorithm* an algorithm that returns, in radix-β, precision-p floating-point arithmetic, an approximation $y_h + y_\ell$ (i.e., an

unevaluated sum of two floating-point numbers) to an exact result $y = f(x)$ (or $f(x_1, x_2)$) such that

- $|y_\ell| \leqslant \frac{1}{2}\,\mathrm{ulp}(y_h)$;
- there exists a small constant C, much smaller than β^p such that

$$|y - (y_h + y_\ell)| < C \cdot \beta^{-2p} \cdot |y_h|\,.$$

When $y = y_h + y_\ell$ exactly, the transformation that generates y_h and y_ℓ from the inputs of f is called an *error-free transform* in the literature.

The unevaluated sum $y_h + y_\ell$ is a particular case (with two terms only) of a floating-point expansion. Several algorithms have been suggested for performing arithmetic on such expansions [144, 153, 78, 25].

In the first part of this chapter, we briefly recall two well-known error-free transforms used later on in the chapter. Then, we analyze an augmented-precision algorithm for the square-root. In the third part, we use that algorithm for designing augmented-precision algorithms for computing $\sqrt{x^2 + y^2}$, where x and y are floating-point numbers. Such a calculation appears in many domains of scientific computing. It is also an important step when computing complex square roots. The naive method—i.e., straightforward implementation of the formula $\sqrt{x^2 + y^2}$—may lead to spurious overflows or underflows. When there are no overflows nor underflows, it is quite accurate (an elementary calculation shows that on a radix-2, precision-p floating-point system, the relative error is bounded by $2^{-p+1} + 2^{-2p}$).

Friedland [57] avoids spurious overflows by computing $\sqrt{x^2 + y^2}$ as $|x| \cdot \sqrt{1 + (y/x)^2}$ if $|x| \geqslant |y|$, and $|y| \cdot \sqrt{1 + (x/y)^2}$ otherwise.

Kahan[1], and Midy and Yakovlev [120] normalize the computation using a power of the radix of the computer system: in radix 2, if $|x| \geqslant |y|$, let b_x be the largest power of 2 less than or equal to x, what they actually compute is $b_x \cdot \sqrt{(x/b_x)^2 + (y/b_x)^2}$. Their solution is less portable (and possibly on some systems, less fast, at least in the context of fixed-precision, hardware-based, arithmetic) than Friedland's solution, yet it will in general be slightly more accurate, since division and multiplication by b_x is exact. Our augmented-precision algorithms will derive from this one. As noticed by Kahan, the IEEE 754 Standard for Floating-Point Arithmetic [85] defines functions `scaleB` and `logB` that make this scaling of x and y easier to implement.

Hull et al. [82] use the naive method along with the exception-handling possibilities specified by the IEEE 754-1985 Standard to recover a correct result when the naive method fails.

In the fourth part, we investigate the possibility of correctly rounding $\sqrt{x^2 + y^2}$ (assuming round-to-nearest). This requires solving the *Table Maker's Dilemma* for that function.

6.2 Two Well-Known Error-Free Transforms

6.2.1 The Fast2Sum Algorithm

The Fast2Sum algorithm was first introduced by Dekker [47], but the three operations of this algorithm already appeared in 1965 as a part of a summation

[1] Unpublished lecture notes

algorithm, called "Compensated sum method," due to Kahan [91]. Under conditions spelled out by Theorem 6.1, it returns the floating term s nearest to a sum $a+b$ and the error term $t = (a+b)-s$. Throughout the chapter, $\mathrm{RN}(u)$ will mean "u rounded to the nearest even", unless otherwise stated (see Section 2.2).

Algorithm 6.1: Fast2Sum [47]

Input: (a, b)
$s \leftarrow \mathrm{RN}(a + b)$
$z \leftarrow \mathrm{RN}(s - a)$
$t \leftarrow \mathrm{RN}(b - z)$
return (s, t)

The following theorem is due to Dekker.

Theorem 6.1 (Fast2Sum algorithm, cf. [47] and [97, Thm. C, p. 236]). *Assume the FP system being used has radix $\beta \leqslant 3$, subnormal numbers available, and provides correct rounding with rounding to nearest.*

Let a and b be FP numbers, and assume that the exponent of a is larger than or equal to that of b (this condition might be difficult to check, but of course, if $|a| \geqslant |b|$, it will be satisfied). Algorithm 6.1 computes two FP numbers s and t that satisfy the following:

- *$s + t = a + b$ exactly;*
- *s is the FP number that is closest to $a + b$.*

We remind the reader that we use as a definition of the exponents the one given page 16 (in Chapter 2).

6.2.2 The TwoMultFMA Algorithm

The FMA instruction makes it possible to evaluate $\pm ax \pm b$, where a, x, and b are floating-point numbers, with one final rounding only. That instruction was introduced in 1990 on the IBM RS/6000. It allows for faster and, in general, more accurate dot products, matrix multiplications, and polynomial evaluations. It also makes it possible to design fast algorithms for correctly-rounded division and square root [113].

The FMA instruction is included in the newly revised IEEE 754-2008 standard for floating-point arithmetic [85].

If an FMA instruction is available, then, to compute the error of a floating-point multiplication $x_1 \cdot x_2$, one can design a very simple algorithm, which only requires two consecutive operations, and works for any radix and precision, provided the product does not overflow and $e_{x_1} + e_{x_2} \geqslant e_{\min} + p - 1$, where e_{x_1} and e_{x_2} are the exponents of x_1 and x_2, and $e_{\min}$ is the minimum exponent of the floating-point system:

6.3 Augmented-Precision Real Square Root with an FMA

Let us now present an augmented-precision real square root algorithm. That algorithm is straightforwardly derived from the following theorem, given in [16]

(see also [128]):

Theorem 6.2 (Computation of square root residuals using an FMA [16]). *Assume x is a precision-p, radix-β, positive floating-point number. If σ is $\sqrt{x}$ rounded to a nearest floating-point number then*

$$x - \sigma^2$$

is exactly computed using one FMA instruction, with any rounding mode, provided that

$$2e_\sigma \geqslant e_{min} + p - 1, \tag{6.1}$$

where e_σ is the exponent of σ.

Notice that similar approximations are used in [112] in a different context (to return a correctly-rounded square root from an accurate enough approximation), as well as in [78] for manipulating floating-point expansions. This is not surprising, since behind this approximation there is nothing but the Taylor expansion of the square-root. What we do claim here, is that we have been able to compute a very tight error bound for Algorithm 6.3 (indeed, an asymptotically optimal one as $p \to +\infty$, as we will see later on). That error bound is given by the following theorem, which shows that the number r returned by Algorithm 6.3 is a very sharp estimate of the error $\sqrt{x} - \sigma$.

Theorem 6.3. *In radix-2, precision-p arithmetic, if the exponent e_x of the FP number x satisfies $e_x \geqslant e_{min}+p$, then the output (σ, r) of Algorithm 6.3 satisfies $\sigma = \mathrm{RN}(\sqrt{x})$ and*

$$\left|(\sigma + r) - \sqrt{x}\right| < 2^{-p-1}\,\mathrm{ulp}(\sigma),$$

and

$$\left|(\sigma + r) - \sqrt{x}\right| < 2^{-2p} \cdot \sigma.$$

(Notice that the second inequality is a straightforward consequence of the first: one give it here for the sake of completeness.)

Proof. First, if $e_x \geqslant e_{\min} + p$ then $x \geqslant 2^{e_{\min}+p}$, so that

$$\sqrt{x} \geqslant 2^{\frac{e_{\min}+p}{2}} \geqslant 2^{\left\lfloor \frac{e_{\min}+p}{2} \right\rfloor},$$

which implies

$$\sigma = \mathrm{RN}(\sqrt{x}) \geqslant 2^{\left\lfloor \frac{e_{\min}+p}{2} \right\rfloor},$$

therefore,

$$e_\sigma \geqslant \left\lfloor \frac{e_{\min} + p}{2} \right\rfloor,$$

Algorithm 6.2: TwoMultFMA

Input: (x_1, x_2)
$r_1 \leftarrow \mathrm{RN}(x_1 \cdot x_2)$
$r_2 \leftarrow \mathrm{RN}(x_1 \cdot x_2 - r_1)$
return (r_1, r_2)

Algorithm 6.3: Augmented computation of $\sqrt{x}$

$\sigma \leftarrow \mathrm{RN}(\sqrt{x})$
$t \leftarrow x - \sigma^2$ `// exact operation through an FMA`
$r \leftarrow \mathrm{RN}(t/(2\sigma))$
return (σ, r)

so that we have,

$$2e_\sigma \geqslant e_{\min} + p - 1.$$

Therefore, Theorem 6.2 applies: $t = x - \sigma^2$ is a floating-point number, so that it is exactly computed using an FMA instruction.

Now, since $\sigma = \mathrm{RN}(\sqrt{x})$ and σ is a normal number (a square root never underflows nor overflows), and since the square root of a floating-point number is never equal to a midpoint [112, 87], we have

$$|\sigma - \sqrt{x}| < 2^{-p} \cdot 2^{e_\sigma},$$

which gives

$$|t| = |\sigma^2 - x| = |\sigma - \sqrt{x}| \cdot |\sigma + \sqrt{x}| < 2^{-p+e_\sigma} \cdot (2\sigma + 2^{e_\sigma - p}).$$

Notice that 2σ is a floating-point number, and that $\mathrm{ulp}(2\sigma) = 2^{e_\sigma - p+2}$. Therefore *there is no floating-point number between* 2σ *and* $2\sigma + 2^{e_\sigma - p}$. Hence, since $|t|/2^{-p+e_\sigma}$ is a floating-point number less than $2\sigma + 2^{e_\sigma - p}$, we obtain

$$|t| \leqslant 2^{-p+e_\sigma+1} \cdot \sigma,$$

implying

$$\left|\frac{t}{2\sigma}\right| \leqslant 2^{-p+e_\sigma}.$$

Also, since 2^{-p+e_σ} is a floating-point number, the monotonicity of the round-to-nearest function implies

$$\left|\mathrm{RN}\left(\frac{t}{2\sigma}\right)\right| \leqslant 2^{-p+e_\sigma}.$$

From these two inequalities, we deduce

$$\left|\mathrm{RN}\left(\frac{t}{2\sigma}\right) - \frac{t}{2\sigma}\right| \leqslant 2^{-2p-1+e_\sigma}. \tag{6.2}$$

Notice (we will need that in Section 6.4) that

$$\left|\mathrm{RN}\left(\frac{t}{2\sigma}\right)\right| \leqslant 2^{-p} \cdot \sigma. \tag{6.3}$$

Now, define a variable ϵ as

$$\sqrt{x} = \sigma + \frac{t}{2\sigma} + \epsilon,$$

where

$$\begin{aligned}\epsilon &= \sqrt{x} - \sigma - \frac{t}{2\sigma} \\ &= \frac{t}{\sqrt{x}+\sigma} - \frac{t}{2\sigma} \\ &= t \cdot \frac{2\sigma - (\sqrt{x}+\sigma)}{(\sqrt{x}+\sigma)\cdot 2\sigma} \\ &= -\frac{(\sigma-\sqrt{x})^2}{2\sigma},\end{aligned}$$

from which we deduce

$$|\epsilon| < \frac{2^{-2p+2e_\sigma}}{2\sigma} \leqslant 2^{-2p-1+e_\sigma}. \tag{6.4}$$

By combining (6.2) and (6.4), we finally get

$$\left|\left(\sigma + \mathrm{RN}\left(\frac{t}{2\sigma}\right)\right) - \sqrt{x}\right| < 2^{-2p+e_\sigma}.$$

This gives an error in ulps as well as a relative error: since $\mathrm{ulp}(\sigma) = 2^{e_\sigma - p + 1}$ we obtain

$$\left|\left(\sigma + \mathrm{RN}\left(\frac{t}{2\sigma}\right)\right) - \sqrt{x}\right| < 2^{-p-1}\,\mathrm{ulp}(\sigma),$$

and

$$\left|\left(\sigma + \mathrm{RN}\left(\frac{t}{2\sigma}\right)\right) - \sqrt{x}\right| < 2^{-2p}\cdot\sigma. \qquad \square$$

Notice that the bound given by Theorem 6.3 is quite tight. Consider as an example the case $p = 24$ (binary32 precision of the IEEE 754-2008 Standard). Assume x is the floating-point number

$$x = 8402801 \cdot 2^{-23} = 1.00169193744659423828125,$$

then one easily gets

$$\sigma = 8395702 \cdot 2^{-23} = 1.000845670700073242187 5,$$

and

$$r = -16749427 \cdot 2^{-48} \approx -5.950591841497 \times 10^{-8},$$

which gives

$$\left|(\sigma + r) - \sqrt{x}\right| = 0.9970012\cdots \times 2^{-48} \times \sigma,$$

to be compared to our bound $2^{-48} \times \sigma$.

Furthermore, the error bounds given by Theorem 6.3 are *asymptotically optimal*, as we can exhibit a family (for p multiple of 3) of input values parametrized by the precision p, such that for these input values, $\left|(\sigma + r) - \sqrt{x}\right| / \sigma$ is asymptotically equivalent to 2^{-2p} as $p \to \infty$. Just consider, for p being a multiple of 6, the input number

$$x = 2^p + 2^{\frac{p}{3}+1} + 2,$$

and for p odd multiple of 3, the input number

$$x = 2^{p-1} + 2^{p/3} + 1.$$

If p is multiple of 6 (the case where p is an odd multiple of 3 is very similar), tedious yet not difficult calculations show that

$$\begin{array}{lcl}
\sqrt{x} & = & 2^{p/2} + 2^{-p/6} + 2^{-p/2} - 2^{-1-5p/6} - 2^{-7p/6} \\
 & & +3 \cdot 2^{-1-11p/6} + \cdots, \\
\sigma & = & 2^{p/2} + 2^{-p/6}, \\
t & = & 2 - 2^{-p/3}, \\
t/(2\sigma) & = & 2^{-p/2} \cdot \left(1 - 2^{-p/3-1} - 2^{-2p/3} + 2^{-p-1}\right. \\
 & & \left. +2^{-4p/3} - \cdots\right), \\
r & = & 2^{-p/2} \cdot \left(1 - 2^{-p/3-1} - 2^{-2p/3} + 2^{-p}\right), \\
\sigma + r & = & 2^{p/2} + 2^{-p/6} + 2^{-p/2} - 2^{-1-5p/6} - 2^{-7p/6} \\
 & & +2^{-3p/2},
\end{array}$$

so that

$$\sqrt{x} - (\sigma + r) \sim_{p\to\infty} 2^{-3p/2} \cdot \left(-1 + 3 \cdot 2^{-1-p/3}\right),$$

from which we derive

$$|\sqrt{x} - (\sigma + r)| \sim_{p\to\infty} 2^{-p-1}\,\mathrm{ulp}(\sigma),$$

and

$$\left|\frac{\sqrt{x} - (\sigma + r)}{\sigma}\right| \sim_{p\to\infty} 2^{-2p}\left(1 - 3 \cdot 2^{-1-p/3}\right),$$

which shows the asymptotic optimality of the bounds given by Theorem 6.3.

6.4 Augmented-Precision 2D Norms

We suggest two very slightly different algorithms. Algorithm 6.5 requires three more operations (a Fast2Sum) than Algorithm 6.4, but it has a slightly better error bound. Again, as for the square-root algorithm, these algorithms derive quite naturally from the Taylor series for the square root: the novelty we believe we bring here is that we provide proven and tight error bounds.

Notice that if one is just interested in getting a very accurate floating-point approximation to $\sqrt{x^2 + y^2}$ (that is, if one does not want to compute the error term r_ℓ), then it suffices to replace the last Fast2Sum instruction by "$r_h \leftarrow \mathrm{RN}(r'_1 + r'_3)$" in both algorithms. Also notice that if the functions `scaleB` and `logB` defined by the IEEE 754-2008 Standard are available and efficiently implemented, one can replace lines 4, 5 and 6 of both algorithms with

$e_x \leftarrow$ `logB`(x)
$\hat{x} \leftarrow$ `scaleB`$(x, -e_x)$
$\hat{y} \leftarrow$ `scaleB`$(y, -e_x)$

and Lines 16 and 17 of Algorithm 6.4, or lines Lines 17 and 18 of Algorithm 6.5 with

$r'_1 \leftarrow$ `scaleB`(r_1, e_x)
$r'_3 \leftarrow$ `scaleB`(r_3, e_x).

Algorithm 6.4: Augmented computation of $\sqrt{x^2+y^2}$

if $|y| > |x|$ **then**
  swap(x, y)
end
$b_x \leftarrow$ largest power of 2 less than or equal to x
$\hat{x} \leftarrow x/b_x$ `// exact operation`
$\hat{y} \leftarrow y/b_x$ `// exact operation`
$(s_x, \rho_x) \leftarrow \text{TwoMultFMA}(\hat{x}, \hat{x})$
$(s_y, \rho_y) \leftarrow \text{TwoMultFMA}(\hat{y}, \hat{y})$
$(s_h, \rho_s) \leftarrow \text{Fast2Sum}(s_x, s_y)$
$s_\ell \leftarrow \text{RN}(\rho_s + \text{RN}(\rho_x + \rho_y))$
$r_1 \leftarrow \text{RN}(\sqrt{s_h})$
$t \leftarrow s_h - r_1^2$ `// exact operation through an FMA`
$r_2 \leftarrow \text{RN}(t/(2r_1))$
$c \leftarrow \text{RN}(s_\ell/(2s_h))$
$r_3 \leftarrow \text{RN}(r_2 + r_1 c)$
$r_1' \leftarrow r_1 \cdot b_x$ `// exact operation`
$r_3' \leftarrow r_3 \cdot b_x$ `// exact operation`
$(r_h, r_\ell) \leftarrow \text{Fast2Sum}(r_1', r_3')$
return (r_h, r_ℓ)

Algorithm 6.5: Slightly more accurate augmented computation of $\sqrt{x^2+y^2}$

if $|y| > |x|$ **then**
  swap(x, y)
end
$b_x \leftarrow$ largest power of 2 less than or equal to x
$\hat{x} \leftarrow x/b_x$ `// exact operation`
$\hat{y} \leftarrow y/b_x$ `// exact operation`
$(s_x, \rho_x) \leftarrow \text{TwoMultFMA}(\hat{x}, \hat{x})$
$(s_y, \rho_y) \leftarrow \text{TwoMultFMA}(\hat{y}, \hat{y})$
$(s_h, \rho_s) \leftarrow \text{Fast2Sum}(s_x, s_y)$
$s_\ell \leftarrow \text{RN}(\rho_s + \text{RN}(\rho_x + \rho_y))$
$(s_h', s_\ell') \leftarrow \text{Fast2Sum}(s_h, s_\ell)$
$r_1 \leftarrow \text{RN}(\sqrt{s_h'})$
$t \leftarrow s_h' - r_1^2$ `// exact operation through an FMA`
$r_2 \leftarrow \text{RN}(t/(2r_1))$
$c \leftarrow \text{RN}(s_\ell'/(2s_h'))$
$r_3 \leftarrow \text{RN}(r_2 + r_1 c)$
$r_1' \leftarrow r_1 \cdot b_x$ `// exact operation`
$r_3' \leftarrow r_3 \cdot b_x$ `// exact operation`
$(r_h, r_\ell) \leftarrow \text{Fast2Sum}(r_1', r_3')$
return (r_h, r_ℓ)

We have the following result

Theorem 6.4 (Accuracy of Algorithms 6.4 and 6.5). *We assume that a radix-2, precision-p (with $p \geqslant 8$), floating-point arithmetic is used and that there are no underflows nor overflows.*

The result (r_h, r_ℓ) returned by Algorithm 6.4 satisfies

$$r_h + r_\ell = \sqrt{x^2 + y^2} + \epsilon,$$

with

$$|\epsilon| < \left(\frac{13}{2} \cdot 2^{-2p} + 31 \cdot 2^{-3p}\right) \cdot |r_h|\,,$$

and

$$|r_\ell| \leqslant \frac{1}{2}\,\mathrm{ulp}(r_h).$$

The result (r_h, r_ℓ) returned by Algorithm 6.5 satisfies

$$r_h + r_\ell = \sqrt{x^2 + y^2} + \epsilon',$$

with

$$|\epsilon'| < \left(\frac{39}{8} \cdot 2^{-2p} + 22 \cdot 2^{-3p}\right) \cdot |r_h|\,,$$

and

$$|r_\ell| \leqslant \frac{1}{2}\,\mathrm{ulp}(r_h).$$

Note that we still don't know if the bounds given by this theorem are sharp.

Proof of Algorithm 6.4. The computations of b_x, $\hat{x}$, and $\hat{y}$ are obviously errorless. We have

$$\begin{aligned} \hat{x}^2 + \hat{y}^2 &= s_x + s_y + \rho_x + \rho_y \\ &= s_h + \rho_s + \rho_x + \rho_y, \end{aligned}$$

with $|\rho_x| \leqslant 2^{-p} s_x$, $|\rho_y| \leqslant 2^{-p} s_y$, and $|\rho_s| \leqslant 2^{-p} s_h$.

We easily find

$$|\rho_x + \rho_y| \leqslant 2^{-p}(s_x + s_y) = 2^{-p}(s_h + \rho_s).$$

Define $u = \mathrm{RN}(\rho_x + \rho_y)$. We have $|u| \leqslant \mathrm{RN}\left(2^{-p}(s_x + s_y)\right)$, so that

$$|u| \leqslant 2^{-p} s_h,$$

and

$$|u - (\rho_x + \rho_y)| \leqslant 2^{-2p} \cdot s_h. \tag{6.5}$$

We therefore get

$$|\rho_s + u| \leqslant 2^{-p+1} \cdot s_h,$$

so that

$$|s_\ell| \leqslant 2^{-p+1} \cdot s_h.$$

Also,

$$|s_\ell - (\rho_s + u)| \leqslant 2^{-2p+1} \cdot s_h.$$

This, combined with (6.5), gives

$$|s_\ell - (\rho_s + \rho_x + \rho_y)| \leqslant 3 \cdot 2^{-2p} \cdot s_h.$$

As a consequence

$$\hat{x}^2 + \hat{y}^2 = s_h + s_\ell + \epsilon_0, \quad \text{with } |\epsilon_0| \leqslant 3 \cdot 2^{-2p} \cdot s_h.$$

Now,

$$\begin{aligned} \sqrt{\hat{x}^2 + \hat{y}^2} &= \sqrt{s_h + s_\ell + \epsilon_0} \\ &= \sqrt{s_h} \cdot \left(1 + \tfrac{s_\ell + \epsilon_0}{2s_h} + \epsilon_1\right), \end{aligned}$$

with (from the Taylor-Lagrange formula)

$$|\epsilon_1| \leqslant \frac{1}{8} \frac{(s_\ell + \epsilon_0)^2}{s_h^2}.$$

From the bounds on s_ℓ and ϵ_0 we get

$$\left|\frac{s_\ell + \epsilon_0}{s_h}\right| \leqslant 2^{-p+1} + 3 \cdot 2^{-2p},$$

which gives

$$|\epsilon_1| \leqslant \frac{1}{8} \cdot \frac{\left(2^{-p+1} + 3 \cdot 2^{-2p}\right)^2}{1 - \left(2^{-p+1} + 3 \cdot 2^{-2p}\right)} < 2^{-2p-1} + 2 \cdot 2^{-3p} \tag{6.6}$$

as soon as $p \geqslant 2$. Hence,

$$\sqrt{\hat{x}^2 + \hat{y}^2} = \sqrt{s_h} \cdot \left(1 + \frac{s_\ell}{2s_h} + \epsilon_2\right) \tag{6.7}$$

with

$$|\epsilon_2| \leqslant |\epsilon_1| + \left|\frac{\epsilon_0}{2s_h}\right| < 2^{-2p+1} + 2 \cdot 2^{-3p}.$$

In Eq. (6.7), $\sqrt{s_h}$ is approximated by $r_1 + r_2$ using Algorithm 6.3. Therefore, from Theorem 6.3, we have

$$\sqrt{s_h} = r_1 + r_2 + \epsilon_3,$$

$$|\epsilon_3| < 2^{-2p} \cdot r_1.$$

Now, $|s_\ell/(2s_h)| \leqslant 2^{-p}$, so that $|c| \leqslant 2^{-p}$ too, and

$$\left|c - \frac{s_\ell}{2s_h}\right| \leqslant 2^{-2p-1}.$$

Hence,

$$\sqrt{\hat{x}^2 + \hat{y}^2} = (r_1 + r_2 + \epsilon_3) \cdot (1 + c + \epsilon_4),$$

with

$$|\epsilon_4| \leqslant \left|c - \frac{s_\ell}{2s_h}\right| + |\epsilon_2| < \frac{5}{2} \cdot 2^{-2p} + 2 \cdot 2^{-3p}.$$

From the bound (6.3) obtained in the proof of Theorem 6.3, we have $|r_2| \leqslant 2^{-p} \cdot |r_1|$. All this gives

$$\sqrt{\hat{x}^2 + \hat{y}^2} = r_1 + r_2 + r_1 c + \epsilon_6,$$

with

$$\begin{aligned} |\epsilon_6| &\leqslant |\epsilon_3| + |r_1\epsilon_4| + |r_2 c| + |r_2\epsilon_4| + |\epsilon_3| \cdot |c + \epsilon_4| \\ &\leqslant r_1 \cdot \left(\frac{9}{2} \cdot 2^{-2p} + \frac{11}{2} \cdot 2^{-3p} + \frac{9}{2} \cdot 2^{-4p} + 2 \cdot 2^{-5p} \right) \\ &\leqslant r_1 \cdot \left(\frac{9}{2} \cdot 2^{-2p} + 7 \cdot 2^{-3p} \right), \end{aligned}$$

as soon as $p \geqslant 2$. Now, from the previously obtained bounds on r_2 and c,

$$|r_2 + r_1 c| \leqslant 2 \cdot 2^{-p} \cdot r_1$$

so that

$$r_3 = r_2 + r_1 c + \epsilon_7,$$

with

$$|\epsilon_7| \leqslant \left(2 \cdot 2^{-2p}\right) \cdot r_1,$$

and (since $2 \cdot 2^{-p} \cdot r_1$ is a floating-point number),

$$|r_3| \leqslant 2 \cdot 2^{-p} \cdot r_1.$$

We therefore conclude that when $b_x = 1$,

$$r_h + r_\ell = r_1 + r_3 = \sqrt{x^2 + y^2} + \epsilon_8,$$

with

$$|\epsilon_8| \leqslant |\epsilon_6| + |\epsilon_7| \leqslant \left(\frac{13}{2} \cdot 2^{-2p} + 7 \cdot 2^{-3p} \right) \cdot r_1.$$

From $r_h + r_\ell = r_1 + r_3$, $|r_\ell| \leqslant 2^{-p}|r_h|$ and $|r_3| \leqslant 2 \cdot 2^{-p} \cdot r_1$, we get

$$|r_1| \leqslant \frac{1 + 2^{-p}}{1 - 2 \cdot 2^{-p}} \cdot |r_h| \leqslant \left(1 + 3 \cdot 2^{-p} + 7 \cdot 2^{-2p}\right) \cdot |r_h|,$$

as soon as $p \geqslant 4$. From this we easily deduce that as soon as $p \geqslant 4$, when $b_x = 1$,

$$|\epsilon_8| \leqslant \left(\frac{13}{2} \cdot 2^{-2p} + 31 \cdot 2^{-3p} \right) \cdot |r_h|.$$

Now, when $b_x \neq 1$, it suffices to notice that r_h, r_ℓ, r_1 and r_2 will be multiplied by the same factor b_x, to deduce that $r_h + r_\ell = \sqrt{x^2 + y^2} + \epsilon_8$, with

$$|\epsilon_8| \leqslant \left(\frac{13}{2} \cdot 2^{-2p} + 31 \cdot 2^{-3p} \right) \cdot |r_h|. \qquad \square$$

Proof of the error bound for Algorithm 6.5. Here again, the computations of b_x, $\hat{x}$, and $\hat{y}$ are obviously errorless. We have

$$\begin{aligned} \hat{x}^2 + \hat{y}^2 &= s_x + s_y + \rho_x + \rho_y \\ &= s_h + \rho_s + \rho_x + \rho_y, \end{aligned}$$

with $|\rho_x| \leqslant 2^{-p} s_x$, $|\rho_y| \leqslant 2^{-p} s_y$, and $|\rho_s| \leqslant 2^{-p} s_h$.

We easily find

$$|\rho_x + \rho_y| \leqslant 2^{-p}(s_x + s_y) = 2^{-p}(s_h + \rho_s)$$

and define $u = \mathrm{RN}(\rho_x + \rho_y)$.

We have $|u| \leqslant \mathrm{RN}\left(2^{-p}(s_x + s_y)\right)$, so that

$$|u| \leqslant 2^{-p} s_h,$$

and

$$|u - (\rho_x + \rho_y)| \leqslant 2^{-2p} \cdot s_h. \tag{6.8}$$

We therefore get

$$|\rho_s + u| \leqslant 2^{-p+1} \cdot s_h,$$

so that

$$|s_\ell| \leqslant 2^{-p+1} \cdot s_h.$$

Also,

$$|s_\ell - (\rho_s + u)| \leqslant 2^{-2p+1} \cdot s_h.$$

This, combined with (6.8), gives

$$|s_\ell - (\rho_s + \rho_x + \rho_y)| \leqslant 3 \cdot 2^{-2p} \cdot s_h,$$

implying that

$$\hat{x}^2 + \hat{y}^2 = s_h + s_\ell + \epsilon_0, \text{ with } |\epsilon_0| \leqslant 3 \cdot 2^{-2p} \cdot s_h.$$

We also have, $s'_h + s'_\ell = s_h + s_\ell$, $|s'_\ell| \leqslant 2^{-p} \cdot s'_h$, and $|s_\ell| \leqslant 2^{-p+1} \cdot s_h$, so that

$$s_h \leqslant \frac{1 + 2^{-p}}{1 - 2^{-p+1}} \cdot s'_h \leqslant (1 + 3 \cdot 2^{-p} + 7 \cdot 2^{-2p}) \cdot s'_h,$$

when $p \geqslant 4$. Which gives

$$\begin{gathered} \hat{x}^2 + \hat{y}^2 = s'_h + s'_\ell + \epsilon_0, \\ \text{with } |\epsilon_0| \leqslant (3 \cdot 2^{-2p} + 10 \cdot 2^{-3p}) \cdot s'_h, \end{gathered}$$

when $p \geqslant 5$.

Now,

$$\begin{aligned} \sqrt{\hat{x}^2 + \hat{y}^2} &= \sqrt{s'_h + s'_\ell + \epsilon_0} \\ &= \sqrt{s'_h} \cdot \left(1 + \tfrac{s'_\ell + \epsilon_0}{2 s'_h} + \epsilon_1\right), \end{aligned}$$

with

$$|\epsilon_1| \leqslant \frac{1}{8} \frac{(s'_\ell + \epsilon_0)^2}{(s'_h)^2}.$$

From the bounds on s'_ℓ and ϵ_0 we get

$$\left|\frac{s'_\ell + \epsilon_0}{s'_h}\right| \leqslant 2^{-p} + 3 \cdot 2^{-2p} + 10 \cdot 2^{-3p},$$

which gives

$$\begin{aligned} |\epsilon_1| &\leqslant \frac{1}{8} \cdot \left(2^{-p} + 3 \cdot 2^{-2p} + 10 \cdot 2^{-3p}\right)^2 \\ &< 2^{-2p-3} + 2^{-3p}, \end{aligned}$$

when $p \geqslant 5$. Hence,

$$\sqrt{\hat{x}^2 + \hat{y}^2} = \sqrt{s'_h} \cdot \left(1 + \frac{s'_\ell}{2s'_h} + \epsilon_2\right) \tag{6.9}$$

with

$$|\epsilon_2| \leqslant |\epsilon_1| + \left|\frac{\epsilon_0}{2s'_h}\right| < \frac{13}{8} \cdot 2^{-2p} + 6 \cdot 2^{-3p}.$$

In Eq. (6.9), $\sqrt{s'_h}$ is approximated by $r_1 + r_2$ using Algorithm 6.3. Therefore, from Theorem 6.3, we have

$$\sqrt{s'_h} = r_1 + r_2 + \epsilon_3,$$

with

$$|\epsilon_3| < 2^{-2p} \cdot r_1.$$

Since $|s'_\ell/(2s'_h)| \leqslant 2^{-p-1}$, so that $|c| \leqslant 2^{-p-1}$ too, and

$$\left|c - \frac{s'_\ell}{2s'_h}\right| \leqslant 2^{-2p-2},$$

hence

$$\sqrt{\hat{x}^2 + \hat{y}^2} = (r_1 + r_2 + \epsilon_3) \cdot (1 + c + \epsilon_4),$$

with

$$|\epsilon_4| \leqslant \left|c - \frac{s'_\ell}{2s'_h}\right| + |\epsilon_2| < \frac{15}{8} \cdot 2^{-2p} + 6 \cdot 2^{-3p}.$$

From the bound (6.3) obtained in the proof of Theorem 6.3, we have $|r_2| \leqslant 2^{-p} \cdot |r_1|$. All this gives

$$\sqrt{\hat{x}^2 + \hat{y}^2} = r_1 + r_2 + r_1 c + \epsilon_6,$$

with

$$\begin{aligned} |\epsilon_6| &\leqslant |\epsilon_3| + |r_1\epsilon_4| + |r_2 c| + |r_2\epsilon_4| + |\epsilon_3| \cdot |c + \epsilon_4| \\ &\leqslant r_1 \cdot \left(\frac{27}{8} \cdot 2^{-2p} + 9 \cdot 2^{-3p}\right), \end{aligned}$$

when $p \geqslant 4$. From the previously obtained bounds on r_2 and c,

$$|r_2 + r_1 c| \leqslant \frac{3}{2} \cdot 2^{-p} \cdot r_1$$

so that

$$r_3 = r_2 + r_1 c + \epsilon_7,$$

with

$$|\epsilon_7| \leqslant \frac{3}{2} \cdot 2^{-2p} \cdot r_1,$$

and

$$|r_3| \leqslant \frac{3}{2} \cdot 2^{-p} \cdot (1 + 2^{-p}) \cdot r_1.$$

We therefore conclude that when $b_x = 1$,

$$r_h + r_\ell = r_1 + r_3 = \sqrt{x^2 + y^2} + \epsilon_8,$$

with

$$|\epsilon_8| \leqslant |\epsilon_6| + |\epsilon_7| \leqslant \left(\frac{39}{8} \cdot 2^{-2p} + 9 \cdot 2^{-3p}\right) \cdot r_1.$$

From $r_h + r_\ell = r_1 + r_3$, $|r_\ell| \leqslant 2^{-p}|r_h|$, and $|r_3| \leqslant \frac{3}{2} \cdot 2^{-p} \cdot (1 + 2^{-p}) \cdot r_1$, we get

$$\begin{aligned} |r_1| &< \frac{1 + 2^{-p}}{1 - \frac{3}{2} \cdot 2^{-p} \cdot (1 + 2^{-p})} \cdot |r_h| \\ &\leqslant \left(1 + \tfrac{5}{2} \cdot 2^{-p} + \tfrac{11}{2} \cdot 2^{-2p}\right) \cdot |r_h|, \end{aligned}$$

when $p \geqslant 6$. From this we finally deduce that when $p \geqslant 8$,

$$|\epsilon_8| \leqslant \left(\frac{39}{8} \cdot 2^{-2p} + 22 \cdot 2^{-3p}\right) \cdot |r_h|.$$

Again, when $b_x \neq 1$, it suffices to notice that r_h, r_ℓ, r_1 and r_2 will be multiplied by the same factor b_x, to deduce that $r_h + r_\ell = \sqrt{x^2 + y^2} + \epsilon_8$, with

$$|\epsilon_8| \leqslant \left(\frac{39}{8} \cdot 2^{-2p} + 22 \cdot 2^{-3p}\right) \cdot |r_h|. \qquad \square$$

6.5 Can We Round $\sqrt{x^2 + y^2}$ Correctly?

In any radix, there are many floating-point values x and y such that $\sqrt{x^2 + y^2}$ is a midpoint [87]. A typical example, in the "toy" binary floating-point system of precision $p = 8$ is $x = 253_{10} = 11111101_2$, $y = 204_{10} = 11001100_2$, for which $\sqrt{x^2 + y^2} = 325_{10} = 101000101_2$.

If the minimum nonzero distance in terms of ulps between $\sqrt{x^2 + y^2}$ and a midpoint is η, then correctly rounding $\sqrt{x^2 + y^2}$ can be done as described in Section 2.4.2.

Hence our purpose in this section is to find lower bounds on the distance between $\sqrt{x^2 + y^2}$ and a midpoint. Notice that in the special case where x and y have the same exponent, Lang and Muller provide similar bounds in [99].

As previously, we assume a binary floating-point arithmetic of precision p. Let x and y be floating-point numbers. Without loss of generality, we assume $0 < y \leqslant x$. Let e_x and e_y be the exponents of x and y. Define $\delta = e_x - e_y$, so $\delta \geqslant 0$. We will now consider two cases.

1. If δ is large

First, let us notice that if x is large enough compared to y, our problem becomes very simple. More precisely, we have

$$\begin{aligned}\sqrt{x^2+y^2} &= x\cdot\sqrt{1+\frac{y^2}{x^2}}\\ &= x+\frac{y^2}{2x}+\epsilon,\end{aligned}$$

with

$$-\frac{1}{8}\frac{y^4}{x^3} < \epsilon < 0.$$

When $y \leqslant 2^{-p/2}x$, we have

$$0 < \frac{y^2}{2x} \leqslant 2^{-p-1}x < \frac{1}{2}\,\mathrm{ulp}(x),$$

so that

$$\left|x-\sqrt{x^2+y^2}\right| = \frac{y^2}{2x}+\epsilon < \frac{1}{2}\,\mathrm{ulp}(x).$$

Hence when $y \leqslant 2^{-p/2}x$, correctly rounding $\sqrt{x^2+y^2}$ is straightforward: it suffices to return x. Notice that $\delta \geqslant p/2+1$ implies $y \leqslant 2^{-p/2}x$. So, let us now focus on the case $\delta < p/2+1$, i.e., $\delta \leqslant \lfloor (p+1)/2 \rfloor$.

2. If δ is small

Since x and y are floating-point numbers, there exist integers M_x, M_y, e_x, and e_y such that

$$\begin{cases} x = M_x\cdot 2^{e_x-p+1}\\ y = M_y\cdot 2^{e_y-p+1},\end{cases}$$

with $0 < M_x, M_y \leqslant 2^p-1$. Assume $\sqrt{x^2+y^2}$ is within ϵ ulps from a midpoint of the form $(M_s+1/2)\cdot 2^{e_s-p+1}$ (with $|\epsilon|$ nonzero and much less than $1/2$). Notice that $x \leqslant s \leqslant \mathrm{RU}(x\sqrt{2})$, so that e_s is e_x or e_x+1. We have

$$\sqrt{M_x^2\cdot 2^{2e_x}+M_y^2\cdot 2^{2e_y}} = \left(M_s+\frac{1}{2}+\epsilon\right)\cdot 2^{e_s},$$

which gives

$$\epsilon = 2^{-e_s}\sqrt{M_x^2\cdot 2^{2e_x}+M_y^2\cdot 2^{2e_y}} - \left(M_s+\frac{1}{2}\right).$$

This implies

$$\begin{aligned}\epsilon &= 2^{-e_s}\cdot\frac{2^{2e_y}\left(M_x^2\cdot 2^{2\delta}+M_y^2\right)-2^{2e_s}\left(M_s^2+M_s+\frac{1}{4}\right)}{2^{e_y}\sqrt{M_x^2\cdot 2^{2\delta}+M_y^2}+2^{e_s}\left(M_s+\frac{1}{2}\right)}\\ &= 2^{-e_s}\frac{N}{D}. \qquad (6.10)\end{aligned}$$

Now since $M_s \leqslant 2^p-1$, $2^{e_s}\cdot\left(M_s+\frac{1}{2}\right)$ is less than 2^{p+e_s} and $|\epsilon| < 1/2$, then $2^{e_y}\sqrt{M_x^2\cdot 2^{2\delta}+M_y^2}$ is less than 2^{p+e_s} too, so that the term D in (6.10) is less than 2^{p+e_s+1}.

Notice that if $e_s = e_x + 1$ we can improve on that bound. In that case,

$$\begin{aligned}\sqrt{M_x^2 \cdot 2^{2e_x} + M_y^2 \cdot 2^{2e_y}} &< \sqrt{2^{2p} + 2^{2p-2\delta}} \cdot 2^{e_x} \\ &= \tfrac{1}{2}\sqrt{1 + 2^{-2\delta}} \cdot 2^{p+e_s},\end{aligned}$$

so that, when $e_s = e_x + 1$,

$$D < \left(\frac{1}{2}\sqrt{1 + 2^{-2\delta}} + 1\right) \cdot 2^{p+e_s}.$$

Let us now focus on the term N in (6.10). It is equal to

$$2^{2e_x-2\delta}\left(M_x^2 \cdot 2^{2\delta} + M_y^2 - 2^{2(e_s-e_x)+2\delta}\left(M_s^2 + M_s + \frac{1}{4}\right)\right),$$

therefore

- if $e_s = e_x+1$ or $\delta > 0$, then N is an integer multiple of $2^{2e_x-2\delta} = 2^{2e_s-2-2\delta}$. Hence, if ϵ is nonzero, its absolute value is at least

$$2^{-e_s} \cdot \frac{2^{2e_s-2-2\delta}}{\left(\frac{1}{2}\sqrt{1 + 2^{-2\delta}} + 1\right) \cdot 2^{p+e_s}} = \frac{2^{-2-2\delta}}{\frac{1}{2}\sqrt{1 + 2^{-2\delta}} \cdot 2^p + 2^p};$$

- if $e_s = e_x$ and $\delta > 0$, then again N is an integer multiple of $2^{2e_s-2\delta}$. Hence, if ϵ is nonzero, its absolute value is at least

$$2^{-e_s} \cdot \frac{2^{2e_s-2\delta}}{2^{p+e_s+1}} \geqslant 2^{-p-1-2\delta}.$$

- if $e_s = e_x$ and $\delta = 0$ then

$$N = 2^{2e_s}\left(M_x^2 + M_y^2 - M_s^2 - M_s - \frac{1}{4}\right)$$

is a multiple of $2^{2e_s}/4$, so that if ϵ is nonzero, its absolute value is at least 2^{-p-3}.

To summarize what we have obtained so far in the case "delta is small", whenever $\epsilon \neq 0$, its absolute value is lower-bounded by

$$2^{-p-3} \tag{6.11}$$

in the case $\delta = 0$; and

$$\min\left\{\frac{2^{-2-2\delta}}{\frac{1}{2}\sqrt{1 + 2^{-2\delta}} \cdot 2^p + 2^p}; 2^{-p-1-2\delta}\right\} = \frac{2^{-p-1-2\delta}}{\sqrt{1 + 2^{-2\delta}} + 2} \tag{6.12}$$

in the case $\delta > 0$.

Now we can merge the various cases considered above and deduce

Theorem 6.5. *If x and y are radix-2, precision-p, floating-point numbers, then*

- *if* $|y| \leqslant 2^{-p/2}\,|x|$ *then* $\mathrm{RN}\left(\sqrt{x^2+y^2}\right) = |x|$*;*
- *if* $|x| \leqslant 2^{-p/2}\,|y|$ *then* $\mathrm{RN}\left(\sqrt{x^2+y^2}\right) = |y|$*;*
- *otherwise, either* $\sqrt{x^2+y^2}$ *is a midpoint, or it is at a distance of at least*

$$\frac{2^{-p-1-2\lfloor (p+1)/2\rfloor}}{2+\sqrt{2}}\,\mathrm{ulp}\left(\sqrt{x^2+y^2}\right)$$

from a midpoint.

When x and y are close, we obtain a much sharper result. For instance, when they are within a factor of 2, δ is equal to 0 or 1, which gives

Theorem 6.6. *If* x *and* y *are radix-2, precision-*p*, floating-point numbers such that* $|x/2| \leqslant |y| \leqslant 2\cdot|x|$*, then either* $\sqrt{x^2+y^2}$ *is a midpoint, or it is at a distance at least*

$$\frac{2^{-p-2}}{\sqrt{5}+4}\,\mathrm{ulp}\left(\sqrt{x^2+y^2}\right)$$

from a midpoint.

Tables 6.1 and 6.2 compare the actual minimum distance to a midpoint (obtained through exhaustive computation) and the bounds we have obtained in this section, in the case of "toy" floating-point systems of precision $p = 10$ and 15 (an exhaustive search was not possible for significantly wider formats). One can see on these tables that in the cases $\delta = 0$ or $\delta = 1$, our bounds are close to the minimum distance (a consequence is that there is little hope of significantly improving the bound given in Theorem 6.6), and that for larger values of δ, our bounds remain of the same order of magnitude as the minimum distance.

6.6 Application: Correct Rounding of $\sqrt{x^2+y^2}$

Various properties can be deduced from the analyses performed in the previous sections. Examples are:

- we can obtain $\sqrt{x^2+y^2}$ correctly rounded in the binary32 format of the IEEE 754-2008 standard if Algorithm 6.4 or Algorithm 6.5 is run in the binary64 format (or a wider format);
- we can obtain $\sqrt{x^2+y^2}$ correctly rounded in the binary64 format of the IEEE 754-2008 standard if Algorithm 6.4 or Algorithm 6.5 is run in the binary128 format;
- if $|x/2| \leqslant |y| \leqslant |2x|$, we can obtain $\sqrt{x^2+y^2}$ correctly rounded in the binary64 format of the IEEE 754-2008 standard if Algorithm 6.4 or Algorithm 6.5 is run in the Intel binary80 format.

However, note that Algorithm 6.4 and Algorithm 6.5 do not necessarily work in presence of underflow (if some variables are subnormal numbers) and overflow. Note also that the proofs developed in Section 6.4 do not guarantee that the algorithms always return the correct rounding of $\sqrt{x^2+y^2}$ in case it is exactly

δ	actual minimum distance to a midpoint	our lower bound to that distance
0	1.23×10^{-4} ulp	1.22×10^{-4} ulp
1	5.72×10^{-5} ulp	3.91×10^{-5} ulp
2	9.49×10^{-5} ulp	1.00×10^{-5} ulp
3	8.76×10^{-6} ulp	2.53×10^{-6} ulp
4	2.01×10^{-6} ulp	6.35×10^{-7} ulp
5	6.24×10^{-7} ulp	1.58×10^{-7} ulp

Table 6.1 – *Comparison between the actual minimum distance to a midpoint (obtained through exhaustive computation) and the bounds obtained in* (6.11) *and* (6.12) *using our method, in the case of a "toy" floating-point system of precision* $p = 10$*. All values in the table are rounded towards* $-\infty$*.*

δ	actual minimum distance to a midpoint	our lower bound to that distance
0	3.81×10^{-6} ulp	3.81×10^{-6} ulp
1	1.71×10^{-6} ulp	1.22×10^{-6} ulp
2	4.65×10^{-7} ulp	3.14×10^{-7} ulp
3	2.38×10^{-7} ulp	7.92×10^{-8} ulp
4	5.96×10^{-8} ulp	1.98×10^{-8} ulp
5	1.49×10^{-8} ulp	4.96×10^{-9} ulp
6	3.76×10^{-9} ulp	1.24×10^{-9} ulp
7	9.85×10^{-10} ulp	3.10×10^{-10} ulp
8	3.81×10^{-5} ulp	7.76×10^{-11} ulp

Table 6.2 – *Comparison between the actual minimum distance to a midpoint (obtained through exhaustive computation) and the bounds obtained in* (6.11) *and* (6.12) *using our method, in the case of a "toy" floating-point system of precision* $p = 15$*. All values in the table are rounded towards* $-\infty$*.*

a midpoint: we know that we get "a rounding-to-nearest," but not necessarily the one specified by the standard tie-breaking rule. Nevertheless, this could probably be achieved by relying on an "exclusion zone" technique, as suggested in Section 2.4.2, using the lower bounds on the minimum nonzero distance from $\sqrt{x^2+y^2}$ to midpoints that we gave in Section 6.5.

6.7 Conclusion

We have given a very tight error bound for a simple augmented-precision algorithm for the square root. We have also introduced two slightly different augmented-precision algorithms for computing $\sqrt{x^2+y^2}$. Then, we have given bounds on the distance between $\sqrt{x^2+y^2}$ and a midpoint, where x and y are floating-point numbers and $\sqrt{x^2+y^2}$ is not a midpoint. These bounds can be used to provide correctly-rounded 2D norms (either using one of our algorithms, or another one).

Chapter 7

Some Issues Related to Double Roundings

The work presented in this chapter is based on a joint work with Jean-Michel Muller and Guillaume Melquiond.

7.1 Double Roundings and Similar Problems

Double rounding is a phenomenon that may occur when different floating-point precisions are available on a same system, or when performing scaled operations whose final result is subnormal. Although double rounding is, in general, innocuous, it may change the behavior of some useful small floating-point algorithms. We analyze the potential influence of double roundings on the Fast2Sum and TwoSum algorithms, as well as on some summation algorithms.

When several floating-point formats are supported in a given environment, it is sometimes difficult to know in which format some operations are performed. This may make the result of a sequence of arithmetic operations somewhat difficult to predict, unless adequate compilation switches are selected. This is an issue addressed by the recent IEEE 754-2008 standard for floating-point arithmetic [85], so the situation might become clearer in the future. However, current users have to face the problem. For instance, the C99 standard states [86, Section 5.2.4.2.2]:

> *the values of operations with floating operands and values subject to the usual arithmetic conversions and of floating constants are evaluated to a format whose range and precision may be greater than required by the type.*

To simplify, assume the various declared variables of a program are of the same format. Two phenomena may occur when a wider format is available in hardware (a typical example is the "double-extended format" available on Intel processors, for variables declared in the double precision/binary64 format):

- for *implicit* variables such as the result of "`a+b`" in the expression "`d = (a+b)*c`" it is not clear in which format they are computed. It may be preferable in most cases to compute them in the wider format;

- *explicit* variables may be first computed in the wider format, and then rounded to their destination format. This sometimes leads to a subtle problem called *double rounding* in the literature. Consider the following C program [128]:

```
double a = 1848874847.0;
double b = 19954562207.0;
double c;
c = a * b;
printf("c = %20.19e\n", c);
return 0;
```

Depending on the processor and the compilation options, we will either obtain

`3.6893488147419103232e+19`

or

`3.6893488147419111424e+19`,

which is the double-precision/binary64 number closest to the exact product. Let us explain this. The exact value of `a*b` is

$$36893488147419107329,$$

whose binary representation is

$$\overbrace{\underbrace{10000000000000000000000000000000000000000000000000000}_{53\text{ bits}}10000000000}^{64\text{ bits}}01$$

If the product is first rounded to the "double-extended precision" format that is available on x87-compatible Intel processors, we get (in binary):

$$\overbrace{\underbrace{10000000000000000000000000000000000000000000000000000}_{53\text{ bits}}10000000000}^{64\text{ bits}}\times 4$$

Then, if that intermediate value is rounded to the double-precision destination format, this gives (using the round-to-nearest-even rounding mode)

$$\underbrace{10000000000000000000000000000000000000000000000000000}_{53\text{ bits}} \times 2^{13}$$
$$= 36893488147419103232_{10},$$

In most applications, these phenomena are innocuous. However, they may make the behavior of some numerical programs difficult to predict (interesting examples are given by Monniaux [124]).

Most compilers offer options that prevent this problem. For instance, on a Linux Debian Etch 64-bit Intel platform, with GCC, the

```
-march=pentium4 -mfpmath=sse
```

compilation switches force the results of operations to be computed and stored in the 64-bit Streaming SIMD Extension (SSE) registers. However, such solutions have drawbacks:

- they significantly restrict the portability of numerical programs: e.g., it is difficult to make sure that one will always use algorithms such as TwoSum or Fast2Sum (see Section 7.1.1), in a large code, with the right compilation switches;
- they may have a bad impact on the performances of programs, as well as on their accuracy, since in most parts of a numerical program, it is in general more accurate to perform the intermediate calculations in a wider format.

Hence, it is of interest to examine which properties of some basic computer arithmetic building blocks remain true when some intermediate operations may be performed in a wider format and/or when double roundings may occur. When these properties suffice, the obtained programs will be much more portable and "robust". Interestingly enough, as shown by Boldo and Melquiond, double roundings could be avoided if the wider precision calculations were implemented in a special rounding mode called *rounding to odd* [18]. Unfortunately, as we are writing this document, rounding to odd is not implemented in floating-point units.

Also, with the four arithmetic operations and the square root, one may rather easily find conditions on the precision of the wider format under which double roundings are innocuous. Such conditions have been made explicit by Figueroa [55, 54] (who mentions in his paper that they probably have been given by Kahan in a course he gave in 1988). For instance, in binary floating-point arithmetic, if the "target" format is of precision $p \geqslant 4$ and the wider format is of precision $p+p'$, double roundings are innocuous for addition if $p' \geqslant p+1$, for multiplication and division if $p' \geqslant p$, and for square root if $p' \geqslant p+2$. Notice that in the most frequent case (namely, $p = 53$ and $p' = 11$) these conditions are not satisfied. For Euclidean division, the problem was addressed by Lefèvre [100].

Double roundings may also cause a problem in binary to decimal conversions. Solutions are given by Goldberg [62], and by Cornea et al [42].

When the rounding mode (or direction) is towards, $+\infty$, $-\infty$ or 0, one may easily check that double roundings cannot change the result of a calculation. As a consequence, in this work, we will focus on "round to nearest" only.

The chapter is organized as follows:

- Section 7.1.1 defines some extra notations, generalizes the standard "ϵ-model" for bounding errors of FP operations with double roundings, and recalls the classical TwoSum algorithm;
- Section 7.2.1 gives some preliminary remarks that will be useful later on in the proofs;
- Sections 7.2.2 and 7.2.3 analyze the behavior of the Fast2Sum and TwoSum algorithms in the presence of double roundings. The main results of that part are Theorems 7.2 and 7.3, which show that even if Fast2Sum or

TwoSum can no longer always return the error of a floating-point addition (because that error is not always exactly representable), they will always return the floating-point number *nearest* that error;

- Fast2Sum and TwoSum are basic building blocks of many summation algorithms. In Section 7.2.4, we give some implications of the results obtained in the previous two sections to the behavior of these algorithms.

- Finally Section 7.3 will be devoted to the formal verification of the mathematical results mentioned in Sections 7.2.1 and 7.2.2, using the Coq formal proof assistant. We will give a detailed outline at the beginning of Section 7.3.

7.1.1 Extra Notations and Background Material

Even, Odd, Normal and Subnormal Numbers, Underflow

We will say that a finite floating-point number is *even* (resp. *odd*) if its integral significand is even (resp. odd).

Concerning underflow, we will follow here the rule for raising the underflow flag of the default exception handling of the IEEE 754-2008 standard [85], and say that an operation induces an underflow when the result is both subnormal and inexact.

Target Format, Wider Internal Format, Roundings

In the sequel, we assume a precision-p target binary format, and a precision-$(p+p')$ wider "internal" format. When we just write that a number x is a floating-point number without explicitly giving its precision, we mean that it is a precision-p FP number. We assume that the set of possible exponents of the wider format contains the set of possible exponents of the target format, and in the following, $e_{\min}$ and $e_{\max}$ denote the extremal exponents of the target format.

$\mathrm{RN}_k(u)$ means u *rounded to the nearest precision-k FP number* (assuming round to nearest *even*: if u is exactly halfway between two consecutive precision-k FP numbers, $\mathrm{RN}_k(u)$ is the one of these two numbers that is even). When k is omitted, it means that $k = p$.

We will also use the notion of *faithful rounding*, presented in Remark 2.3 on page 21.

We also say that a number x *fits in k bits* if it is equal to a precision-k FP number, or, equivalently, if in the bit string $\mathcal{S}$ constituted by the binary representation of x there is a chain of at most k consecutive bits that contains all the nonzero bits of $\mathcal{S}$.

Midpoints

We recall that a *precision-p midpoint* is a number exactly halfway between two consecutive precision-p FP numbers, and that these numbers enjoy the following properties:

Theorem 7.1 (On midpoints). *If x and y are real numbers such that $x \neq y$, and if there is no midpoint between x and y, then we have*

$$\mathrm{RN}(x) = \mathrm{RN}(y).$$

Moreover, in any radix-2 FP format, we have:

- *if x is a nonzero FP number, and if $|x|$ is not a power of 2, then the two midpoints that surround x are $x - \frac{1}{2}\,\mathrm{ulp}(x)$ and $x + \frac{1}{2}\,\mathrm{ulp}(x)$;*
- *if $|x|$ is a power of 2 strictly larger than $2^{e_{min}}$ and less than or equal to $2^{e_{max}}$ then the two midpoints that surround x are $x - \mathrm{sign}(x) \cdot \frac{1}{4}\,\mathrm{ulp}(x)$ and $x + \mathrm{sign}(x) \cdot \frac{1}{2}\,\mathrm{ulp}(x)$.*

Double Roundings and Double Rounding Slips

In the literature, the term "double rounding" either just means that two roundings occurred, or means that two roundings occurred *and* that this changed the result. To distinguish between these two events, we will say that, when the arithmetic operation $x \top y$ appears in a program:

- a *double rounding* occurs if what is actually performed is

$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(x \top y)\right),$$

- a *double rounding slip* occurs if a double rounding occurs and the obtained result differs from $\mathrm{RN}_p(x \top y)$.

(a very similar definition can be given for an unary function such as $\sqrt{x}$).

The "standard model", or "ϵ-model"

In the sequel, we will often use the following property that is very similar to the usual ϵ-model that we recalled in Property 2.1 (on page 20). Specifically, it can be shown as a corollary of Property 2.1, by using the fact that two consecutive roundings, one in precision $p + p'$, and one in precision p, are performed:

Property 7.1 (ϵ-model with double roundings). *Let a and b be precision-p FP numbers, and let $\top \in \{+, -, \times, \div\}$.*

- *if no underflow nor overflow occurs, then*

$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a \top b)\right) = (a \top b) \cdot (1 + \epsilon_1) = \frac{a \top b}{1 + \epsilon_2}, \tag{7.1}$$

  where $|\epsilon_1|$, $|\epsilon_2| \leqslant \mathbf{u}'$, with $\mathbf{u}' = 2^{-p} + 2^{-p-p'} + 2^{-2p-p'}$;

- *if the result of a floating-point **addition** $a + b$ has absolute value less than $2^{e_{min}}$, then*

$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a + b)\right) = (a + b)$$

  exactly, which implies that for $\top = +$, (7.1) always holds, unless overflow occurs.

Since (when p' is large enough) the bound $\mathbf{u}'$ is only slightly larger than $\mathbf{u}$, most properties that can be shown using the ϵ-model only will remain true in the presence of double roundings (possibly with somewhat larger error bounds).

u, θ_k and γ_k Notations

In [79, page 67], Higham defines notations θ_k and γ_k that turn out to be very useful in error analysis. We will very slightly adapt them to the context of double roundings.

Define $\mathbf{u} = 2^{-p}$ and $\mathbf{u}' = 2^{-p} + 2^{-p-p'} + 2^{-2p-p'}$. For any integer k, θ_k will denote a quantity of absolute value bounded by

$$\gamma_k = \frac{k\,\mathbf{u}}{1 - k\,\mathbf{u}},$$

and θ'_k will denote a quantity of absolute value bounded by

$$\gamma'_k = \frac{k\,\mathbf{u}'}{1 - k\,\mathbf{u}'}.$$

The TwoSum Algorithm

As recalled by Theorem 6.1 on page 133, the Fast2Sum algorithm works in radix $\beta \leqslant 3$ and assumes its FP arguments a and b satisfy $e_a \geqslant e_b$.

When we do not know in advance whether $e_a \geqslant e_b$ or not (or when the radix β is not 2, this latter case being not dealt with in this work), it may be preferable to use the following algorithm, due to Knuth [97] and Møller [123]:

Algorithm 7.1: TwoSum

Input: (a, b)
$s \leftarrow \mathrm{RN}(a + b)$
$a' \leftarrow \mathrm{RN}(s - b)$
$b' \leftarrow \mathrm{RN}(s - a')$
$\delta_a \leftarrow \mathrm{RN}(a - a')$
$\delta_b \leftarrow \mathrm{RN}(b - b')$
$t \leftarrow \mathrm{RN}(\delta_a + \delta_b)$
return (s, t)

Knuth shows that, if a and b are normal FP numbers, then for any radix β, provided that no underflow nor overflow occurs, $a + b = s + t$. Boldo et al. [17] give a formal proof of this algorithm in radix 2, and show that underflow does not hinder the result.

7.2 Mathematical Setup

7.2.1 Some Preliminary Remarks

Let us first notice something about Fast2Sum.

Remark 7.1. The proof of Fast2Sum (see for instance the one given in [128]) relies on the fact that if we have

$$s = \mathrm{RN}(a + b),$$

then the variables z and t of Algorithm Fast2Sum are computed exactly (i.e., $s - a$ and $b - z$ are FP numbers). This implies that the same result will be obtained if these variables are computed in a wider format, or with double

roundings (or with a directed rounding mode). Incidentally, this shows (at least in common languages) that an explicit declaration of variable z is unnecessary, and that in a program, one may safely replace

```
z = s-a; t = b-z
```

with

```
t = b-(s-a).
```

Generally speaking, it is well known that unless overflow occurs, the error of a rounded-to-nearest floating-point addition of two precision-p numbers is a precision-p number: it is precisely that error that is computed by Algorithms 6.1 and 7.1. When double roundings slips occur, the results of sums are very slightly different from rounded to nearest sums. This difference, although it is very small, sometimes suffices to make the error not representable.

More precisely, assume a double rounding slip occurs when evaluating the sum s of two precision-p FP numbers a and b, i.e.,

$$s = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a+b)) \neq \mathrm{RN}_p(a+b).$$

Then if $p' \geqslant 1$ and $p' \leqslant p$, the error $r = a+b-s$ of that floating-point addition may not be exactly representable in precision-p arithmetic.

Proof. To show this, it suffices to consider

$$a = 1\underbrace{xxxx\cdots x}_{p-3 \text{ bits}}01,$$

where $xxxx\cdots x$ is any $(p-3)$-bit bit-chain. The number a is a p-bit integer, thus exactly representable in precision-p FP arithmetic. Also consider

$$b = 0.0\underbrace{111111\cdots 1}_{p \text{ ones}} = \frac{1}{2} - 2^{-p-1}.$$

The number b is equal to $(2^p-1)\cdot 2^{-p-1}$, hence it is a precision-p FP number too.

We thus have:

$$a+b = \underbrace{1xxxx...x01}_{p \text{ bits}}.0\underbrace{111111...1}_{p \text{ bits}},$$

so that if $1 \leqslant p' \leqslant p$,

$$u = \mathrm{RN}_{p+p'}(a+b) = 1xxxx...x01.100...0.$$

The "round to nearest even" rule thus implies

$$s = \mathrm{RN}_p(u) = 1xxxx...x10 = a+1.$$

Therefore,

$$s-(a+b) = a+1-(a+\frac{1}{2}-2^{-p-1}) = \frac{1}{2}+2^{-p-1} = 0.\underbrace{10000\cdots 01}_{p+1 \text{ bits}},$$

which is not exactly representable in precision-p FP arithmetic. □

Remark 7.2. Let a and b be precision-p FP numbers, and define

$$s = \mathrm{RN}_p\Big(\mathrm{RN}_{p+p'}(a+b)\Big)$$

for a given $p' \geqslant 0$. If the exponents of a and b satisfy $e_b \leqslant e_a$ and s is nonzero, then its exponent satisfies

$$e_s \leqslant 1 + e_a. \tag{7.2}$$

Proof. Since $|a|$ and $|b|$ are less than or equal to $(2^p-1)\cdot 2^{e_a-p+1}$, we have

$$|a+b| \leqslant (2^p-1)\cdot 2^{e_a-p+2},$$

so that, given that roundings are monotonically-increasing functions and that $(2^p-1)\cdot 2^{e_a-p+2}$ is a precision-p (and precision-$(p+p')$) FP number, we have

$$|s| = \Big|\mathrm{RN}_p\Big(\mathrm{RN}_{p+p'}(a+b)\Big)\Big| \leqslant (2^p-1)\cdot 2^{(e_a+1)-p+1},$$

therefore $e_s \leqslant e_a + 1$. □

Remark 7.3. Assume we compute $w = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(u+v))$, where u and v are precision-p, radix-2, FP numbers of exponents e_u and e_v, with $e_u \geqslant e_v$. If $p' \geqslant 2$ and a double-rounding slip occurs in that computation, then

$$e_u - p - 1 \leqslant e_v \leqslant e_u - p', \tag{7.3}$$

and

$$e_w \geqslant e_v + p' + 1. \tag{7.4}$$

Proof of (7.3). First, if $e_u - p - 1 > e_v$ (i.e., $e_u - p - 2 \geqslant e_v$), then

$$|v| < 2^{e_v+1} \leqslant 2^{e_u-p-1} = \frac{1}{4}\,\mathrm{ulp}(u). \tag{7.5}$$

Also, $p' \geqslant 2$ implies that $u - \frac{1}{4}\,\mathrm{ulp}(u)$ and $u + \frac{1}{4}\,\mathrm{ulp}(u)$ are precision-$(p+p')$ FP numbers. Therefore,

$$u - \frac{1}{4}\,\mathrm{ulp}(u) \leqslant \mathrm{RN}_{p+p'}(u+v) \leqslant u + \frac{1}{4}\,\mathrm{ulp}(u).$$

Therefore,

- if $|u|$ is not a power of 2 then $\mathrm{RN}_{p+p'}(u+v)$ cannot be a precision-p midpoint. The final result follows from Remark 7.5 below;
- if $|u| = 2^{e_u}$ exactly, then $|u| - \frac{1}{4}\,\mathrm{ulp}(u)$ is a midpoint (it is the only one between $|u| - \frac{1}{4}\,\mathrm{ulp}(u)$ and $|u| + \frac{1}{4}\,\mathrm{ulp}(u)$), but $e_v \leqslant e_u - p - 2$ implies $|v| < 2^{e_u-p-1} = \frac{1}{4}\,\mathrm{ulp}(u)$, so that the real value of $u+v$ is between the midpoint $\mu = u - \mathrm{sign}(u)\cdot\frac{1}{4}\,\mathrm{ulp}(u)$ and $\mu' = u + \mathrm{sign}(u)\cdot\frac{1}{4}\,\mathrm{ulp}(u)$. Given that
$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(\mu)\right) = u \tag{7.6}$$
(due to the round-to-nearest *even* rounding rule), and that
$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(\mu')\right) = u,$$

we can use the fact that the roundings are monotonic functions to deduce that

$$\begin{aligned} \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(u+v)\right) &= u \\ &= \mathrm{RN}(u+v), \end{aligned} \tag{7.7}$$

so that no double rounding slip occurs.

Second, if $e_v > e_u - p'$, then $u+v$ can be written $2^{e_v-p+1}\left(2^{e_u-e_v}M_u + M_v\right)$, where M_u and M_v are the integral significands of u and v, and the integer $2^{e_u-e_v}M_u + M_v$ satisfies

$$\left|2^{e_u-e_v}M_u + M_v\right| \leqslant 2^{p'-1}(2^p-1) + (2^p-1) \leqslant 2^{p+p'} - 1.$$

Therefore $u+v$ is exactly representable in precision $p+p'$, so that no double rounding slip can occur. □

Proof of (7.4). Let k be the integer such that $2^k \leqslant |u+v| < 2^{k+1}$. The monotonicity of the rounding functions implies that

$$2^k \leqslant |\mathrm{RN}_p(\mathrm{RN}_{p+p'}(u+v))| \leqslant 2^{k+1},$$

therefore, e_w is equal to k or $k+1$. Since $u+v$ does not fit into $p+p'$ bits (otherwise there would not be a double rounding slip) and $u+v$ is a multiple of 2^{e_v-p+1}, we deduce that $\mathrm{ulp}_{p+p'}(u+v) > 2^{e_v-p+1}$, which implies that $k-p-p'+1 > e_v-p+1$. Therefore $e_w \geqslant k > e_v+p'$, which concludes the proof. □

Notice that the condition "$p' \leqslant p$" in Remark 7.1 is necessary. More precisely,

Remark 7.4. If $p' \geqslant p+1$ then a double rounding slip cannot occur when computing $a+b$, i.e., we always have

$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a+b)\right) = \mathrm{RN}_p(a+b).$$

Proof. This is a classical result [55, 54]. A sketch of the proof is the following. Assume, without l.o.g., that $|a| \geqslant |b|$: if $e_b \geqslant e_a - p - 1$ then $a+b$ fits in at most $2p+1$ bits, so that as soon as $p' \geqslant p+1$, $\mathrm{RN}_{p+p'}(a+b) = a+b$ exactly; and if $e_b < e_a - p - 1$, then Equation (7.3) in Remark 7.3 implies that no double rounding slip can occur. □

Remark 7.5. Assume, $p' \geqslant 1$, if a double rounding slip occurs when evaluating $a \top b$ (where $\top$ is any operation) then $\mathrm{RN}_{p+p'}(a \top b)$ is a *precision-p midpoint*, i.e., a number exactly halfway between two consecutive precision-p FP numbers.

Proof. The proof of Remark 7.5 is common arithmetic folklore, and just uses the fact that roundings are monotonic functions. Let us give it anyway for the sake of completeness. If $\mathrm{RN}_p(a \top b) \neq \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a \top b))$ then there is a precision-p midpoint, say μ, between $a \top b$ and $\mathrm{RN}_{p+p'}(a \top b)$. That number satisfies

$$|(a \top b) - \mu| \leqslant |(a \top b) - \mathrm{RN}_{p+p'}(a \top b)|\,.$$

μ fits in $(p+1)$ bits. Since $p' \geqslant 1$, it is a precision-$(p+p')$ FP number. Since by definition $\mathrm{RN}_{p+p'}(a \top b)$ is a a precision-$(p+p')$ FP number nearest $a \top b$, we have:

- either there is only one precision-$(p+p')$ FP number nearest $a \top b$—i.e., $a \top b$ is not a precision-$(p+p')$ midpoint, in such a case we necessarily have $\mathrm{RN}_{p+p'}(a \top b) = \mu$;
- or $a \top b$ is a precision-$(p+p')$ midpoint. In such a case, if $\mathrm{RN}_{p+p'}(a \top b) \neq \mu$, then either μ is above $a \top b$ and $\mathrm{RN}_{p+p'}(a \top b)$ is below $a \top b$, or μ is below $a \top b$ and $\mathrm{RN}_{p+p'}(a \top b)$ is above $a \top b$: in any case, μ cannot be between $a \top b$ and $\mathrm{RN}_{p+p'}(a \top b)$, which is a contradiction. □

An immediate consequence of Remark 7.5 (due to the round-to-nearest-even rule) is the following.

Remark 7.6. Assume, $p' \geqslant 1$, if a double rounding slip occurs when evaluating $a \top b$ then the returned result $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(a \top b))$ is an even FP number.

In our proofs, we will also frequently use the following, well-known, result.

Remark 7.7 (Sterbenz Lemma [161]). If a and b are positive FP numbers, and

$$\frac{a}{2} \leqslant b \leqslant 2a,$$

then $a - b$ is a floating-point number, which implies that it will be computed exactly, whatever the rounding.

Finally, the following result will be used later on to prove that even when the error of a floating-point addition is not exactly representable because of a double rounding slip, as soon as $p' \geqslant 2$, we are anyway able to compute the floating-point number nearest that error.

Remark 7.8. Let a and b be precision-p FP numbers, and define

$$s = \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}\left(a+b\right)\right).$$

The number $r = a + b - s$ fits in at most $p+2$ bits, so that as soon as $p' \geqslant 2$, we have

$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}\left(a+b-s\right)\right) = \mathrm{RN}_p(a+b-s). \tag{7.8}$$

Proof. Without l.o.g., we assume $e_a \geqslant e_b$. First, we already know that if no double rounding slip occurs when computing s, namely if $\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}\left(a+b\right)\right) = \mathrm{RN}_p(a+b)$, then $a + b - s$ is a precision-p FP number. In such a case, (7.8) is obviously true. So let us assume that $\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}\left(a+b\right)\right) \neq \mathrm{RN}_p(a+b)$. Equation (7.3) in Remark 7.3 implies therefore that $e_b \geqslant e_a - p - 1$.

Since a and b are both multiple of 2^{e_b-p+1}, we easily deduce that $a + b - s$ too is a multiple of 2^{e_b-p+1}. Moreover by Remark 7.2, we have $e_s \leqslant e_a + 1$.

From all this, we deduce that $a + b - s$ is a multiple of 2^{e_b-p+1} of absolute value less than or equal to

$$2^{-p+e_s} + 2^{-p-p'+e_s} \leqslant 2^{-p+e_a+1} + 2^{-p-p'+e_a+1} \leqslant 2^{e_b+2} + 2^{e_b-p'+2}.$$

Hence, $r = a + b - s$ fits in at most $p+2$ bits, therefore, as soon as $p' \geqslant 2$,

$$\mathrm{RN}_{p+p'}(a+b-s) = a + b - s.$$

□

7.2.2 Behavior of Fast2Sum in the Presence of Double Roundings

Remark 7.1 implies that Algorithms Fast2Sum and TwoSum cannot always return the exact value of the error when the addition $\mathrm{RN}(a+b)$ is replaced by

$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a+b)\right),$$

i.e., when a double rounding occurs, because that error is not always exactly equal to a floating-point number.

And yet, we may try to bound the difference between the exact error and the returned number t (indeed, we will prove that t is the FP number nearest the exact error). Let us analyze how algorithm Fast2Sum behaves when double roundings are allowed. Assume $e_a \geqslant e_b$, we will consider that what is actually performed is

Algorithm 7.2: Fast2Sum-with-double-roundings

Input: (a, b)
$s \leftarrow \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a+b)\right)$
$z \leftarrow \circ(s-a)$
$t \leftarrow \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(b-z)\right)$ or $\mathrm{RN}_p(b-z)$
return (s, t)

where $\circ(u)$ means either $\mathrm{RN}_p(u)$, $\mathrm{RN}_{p+p'}(u)$, or $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(u))$, or any faithful rounding—this is not important, as we will see that $s-a$ is exactly representable in precision-p FP arithmetic, so that it will be computed exactly, whatever the rounding: this means that in a program, one may safely replace

```
z = s-a; t = b-z
```

by

```
t = b-(s-a).
```

Define e_a, e_b, and e_s as the floating-point exponents of a, b, and s, and M_a, M_b, and M_s as their significands. We assume $e_a \geqslant e_b$ (that condition will be satisfied if $|a| \geqslant |b|$). By Remark 7.2, this implies $e_s \leqslant e_a + 1$.

Without l.o.g., we assume $s \geqslant 0$ (otherwise if suffices to change the signs of a and b). We have

$$a = M_a \cdot 2^{e_a - p + 1},$$

with $|M_a| \leqslant 2^p - 1$, and similar relations for b and s. Also, notice that

$$2^{p-1} \leqslant \frac{s}{2^{e_s - p + 1}} \leqslant 2^p - 1$$

implies

$$2^{p-1} - \frac{1}{4} \leqslant \frac{\mathrm{RN}_{p+p'}(a+b)}{2^{e_s - p + 1}} < 2^p - \frac{1}{2},$$

which implies

$$2^{p-1} - \frac{1}{4} - 2^{-p'-2} \leqslant \frac{a+b}{2^{e_s - p + 1}} < 2^p - \frac{1}{2} + 2^{-p'-1}.$$

1. **if** $e_s = e_a + 1$, define $\delta = e_a - e_b$. We have,

$$a + b = 2^{e_s - p + 1} \left(\frac{M_a}{2} + \frac{M_b}{2^{\delta+1}} \right),$$

from which we deduce

$$\mathrm{RN}_{p+p'}(a + b) = 2^{e_s - p + 1} \left(\frac{M_a}{2} + \frac{M_b}{2^{\delta+1}} + \epsilon \right),$$

where $|\epsilon| \leqslant 2^{-p'-1}$. Therefore, we have

$$M_s = \left\lceil \frac{M_a}{2} + \frac{M_b}{2^{\delta+1}} + \epsilon \right\rfloor, \tag{7.9}$$

where $\lceil u \rfloor$ is the integer nearest to u (with round-to-even choice in case of a tie). Now, define $\mu = 2M_s - M_a$. We have,

$$\frac{M_b}{2^\delta} - 1 - 2^{-p'} \leqslant \mu \leqslant \frac{M_b}{2^\delta} + 1 + 2^{-p'}.$$

Since μ is an integer, $\delta \geqslant 0$, and $|M_b| \leqslant 2^p - 1$, if $p' \geqslant 1$, then either $|\mu| \leqslant 2^p - 1$, or $\mu = \pm 2^p$. In both cases, since $s - a = \mu \cdot 2^{e_a - p + 1}$, $s - a$ is exactly representable in precision p.

2. **if** $e_s \leqslant e_a$, define $\delta_1 = e_a - e_b$. We have

$$a + b = \left(2^{\delta_1} M_a + M_b\right) \cdot 2^{e_b - p + 1}.$$

- if $e_s \leqslant e_b$ then $s = a + b$ exactly, since s is obtained by rounding $a + b$ first to the nearest multiple of $2^{e_s - p - p' + 1}$—or to the nearest multiple of $2^{e_s - p - p'}$ in case $|a+b|$ is less than $2^{e_s} - 2^{e_s - p - p' - 1}$—which is a divisor of $2^{e_b - p + 1}$, and then to the nearest multiple of $2^{e_s - p + 1}$ (which is a divisor of $2^{e_b - p + 1}$ too). These two rounding operations left it unchanged since it is already a multiple of $2^{e_b - p + 1}$. Hence in the case $e_s \leqslant e_b$, $s - a = b$ is exactly representable;
- if $e_s > e_b$, let us define $\delta_2 = e_s - e_b$. We have,

$$s = \left\lceil 2^{\delta_1 - \delta_2} M_a + 2^{-\delta_2} M_b + \epsilon \right\rfloor \cdot 2^{e_s - p + 1},$$

where $|\epsilon| \leqslant 2^{-p'-1}$. This implies

$$|s - a| \leqslant \left(2^{-\delta_2} M_b + \frac{1}{2} + 2^{-p'-1} \right) \cdot 2^{e_s - p + 1}.$$

Also, since $e_s \leqslant e_a$, $s - a$ is a multiple of $2^{e_s - p + 1}$. We therefore have,

$$s - a = K \cdot 2^{e_s - p + 1},$$

where K is an integer satisfying

$$|K| \leqslant 2^{-\delta_2} |M_b| + \frac{1}{2} + 2^{-p'-1} < 2^{p-1} + 1 < 2^p - 1.$$

(as soon as $p \geqslant 3$ and $p' \geqslant 1$, which holds in all cases of practical interest). Hence, $s - a$ is exactly representable in precision-p floating-point arithmetic.

We have shown that in all cases, $s-a$ is exactly representable in precision-p FP arithmetic. Hence, variable z of the algorithm will be exactly computed (and the way it is rounded—correct rounding to precision p, correct rounding to an "extended", precision-$(p+p')$ format, double rounding, directed rounding—has no influence on this). Now, from $z = s - a$, we immediately deduce $b - z = (a+b) - s = r$, so that

$$t = \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(r)\right).$$

Using Remark 7.8, we immediately deduce

$$t = \mathrm{RN}_p\left(r\right).$$

In other words, each time r is exactly representable (which happens every time a double rounding slip does not occur when computing $a+b$), we get it exactly; and when r is not exactly representable, we get the precision-p FP number nearest to r. The following theorem summarizes the obtained results.

Theorem 7.2. *Assume a binary target floating-point format of precision $p \geqslant 3$, assume a binary format of precision $p+p'$, with $p' \geqslant 2$, is available. If a and b are precision-p numbers, with $e_a \geqslant e_b$ (that condition will be satisfied if $|a| \geqslant |b|$), and if no overflow occurs, then the sequence of calculations*

$s \leftarrow \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a+b)\right)$
$z \leftarrow \circ(s-a)$
$t \leftarrow \odot(b-z)$

(where $\circ(u)$ means either $\mathrm{RN}_p(u)$, $\mathrm{RN}_{p+p'}(u)$, or $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(u))$, and $\odot(u)$ means either $\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(u)\right)$ or $\mathrm{RN}_p(u)$) satisfies the following property:

- *$z = s - a$ exactly (this will be useful later on);*
- *if no double rounding slip occurred when computing s (in other words, if $s = \mathrm{RN}_p(a+b)$), then $t = (a+b-s)$ exactly;*
- *otherwise, $t = \mathrm{RN}_p(a+b-s)$.*

7.2.3 Behavior of TwoSum in the Presence of Double Roundings

The following preliminary lemma is directly adapted from Lemma 4 in Shewchuk's paper [153].

Lemma 7.1. *Let a and b be precision-p binary floating-point numbers. Let*

$$s = \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a+b)\right) \text{ or } \mathrm{RN}_p(a+b).$$

If $|s| < \min\{|a|, |b|\}$ then $s = a+b$ (that is, s is computed exactly).

Proof. Define $\sigma = a+b$. Without loss of generality, assume that $\min\{|a|,|b|\} = |b|$. Define e_b as the exponent of b and M_b as its integral significand (i.e., $b = M_b \cdot 2^{e_b-p+1}$). Since a and b are both multiples of 2^{e_b-p+1}, σ is a multiple of 2^{e_b-p+1} too. Also, due to the monotonicity of rounding, $|s| < |b|$ implies $|\sigma| < |b|$. An immediate consequence is that $\sigma/2^{e_b-p+1}$ is an integer of absolute value less than $|M_b| \leqslant 2^p - 1$. This implies that σ is a precision-p floating-point number. Therefore $\mathrm{RN}_{p+p'}(\sigma) = \sigma$, and $\mathrm{RN}(\sigma) = \sigma$, so that $s = \sigma$. □

We will analyze the following algorithm

Algorithm 7.3: TwoSum-with-double-roundings

Input: (a, b)

1 $s \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a+b))$ or $\mathrm{RN}_p(a+b)$
2 $a' \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(s-b))$ or $\mathrm{RN}_p(s-b)$
3 $b' \leftarrow \circ(s-a')$
4 $\delta_a \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a-a'))$ or $\mathrm{RN}_p(a-a')$
5 $\delta_b \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(b-b'))$ or $\mathrm{RN}_p(b-b')$
6 $t \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(\delta_a+\delta_b))$ or $\mathrm{RN}_p(\delta_a+\delta_b)$
7 **return** (s, t)

where $\circ(u)$ is either $\mathrm{RN}_p(u)$, $\mathrm{RN}_{p+p'}(u)$, or $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(u))$, or any faithful rounding (indeed, we will show that $s-a'$ is a precision-p FP number, so that b' will be computed exactly).

First, let us raise the following point:

Remark 7.9. Assuming $p' \geqslant 2$, if the variables a' and b' of Algorithm 7.3 satisfy $a' = a$ and $b' = s - a'$ exactly, then $t = \mathrm{RN}(a+b-s)$.

Proof. If $a' = a$ then $\delta_a = 0$. Also, $b' = s - a' = s - a$ implies $b - b' = a + b - s$, so that $\delta_b = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a+b-s))$. This gives

$$t = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a+b-s)) = \mathrm{RN}_p(a+b-s),$$

using Remark 7.8. □

Let us now analyze Algorithm 7.3.

Behavior of Algorithm 7.3 in the case $|b| \geqslant |a|$.
In the case $|b| \geqslant |a|$, Lines 1, 2 and 4 of Algorithm 7.3 constitute Fast2Sum-with-double-roundings(b, a), i.e., Algorithm 7.2, called with input values (b, a). A consequence of this (see Theorem 7.2) is that the computation of Line 2 is exact, which implies $a' = s - b$ and $\delta_a = \mathrm{RN}_p(a+b-s)$. Also, $a' = s - b$ implies $s - a' = b$, so that $b' = b$ exactly and $\delta_b = 0$. All this implies

$$t = \mathrm{RN}_p(a+b-s).$$

Behavior of Algorithm 7.3 in the case $|b| < |a|$ and $|s| < |b|$.
If $|b| < |a|$ and $|s| < |b|$, then Lemma 7.1 applies: $s = a+b$ exactly, which implies $a' = a$, $b' = b$, and $\delta_a = \delta_b = t = 0$.

Behavior of Algorithm 7.3 in the case $|b| < |a|$ and $|s| \geqslant |b|$.
Let us now assume $|b| < |a|$ and $|s| \geqslant |b|$. These inequalities have two important consequences:

- we have $s = (a+b)\cdot(1+\epsilon_1)$ and $a' = (s-b)\cdot(1+\epsilon_2)$, with $|\epsilon_1|, |\epsilon_2| \leqslant \mathbf{u}'$ (we remind the reader that $\mathbf{u}' = 2^{-p} + 2^{-p-p'} + 2^{-p-2p'}$, see Section 7.1.1), from which we easily deduce

$$a' = (a + a\epsilon_1 + b\epsilon_1)\cdot(1+\epsilon_2) = a\cdot(1+\epsilon_3),$$

with $|\epsilon_3| \leqslant 3\,\mathbf{u}' + 2\,\mathbf{u}'^2$. An immediate consequence is that as soon as $p \geqslant 4$ and $p' \geqslant 1$ (which holds in all practical cases), $|a/2| \leqslant |a'| \leqslant |2a|$, and a and a' have the same sign. Hence, from Sterbenz lemma (Remark 7.7), $a - a'$ is a precision-p floating-point number, which implies $\delta_a = a - a'$ exactly. Also (which will be useful later on), $e'_a \leqslant e_a + 1$.

- Lines 2, 3 and 5 constitute Fast2Sum (with double roundings, i.e., Algorithm 7.2), called with input values $(s, -b)$. This implies that $b' = s - a'$ exactly, and $\delta_b = \mathrm{RN}_p(a' - (s - b))$. Moreover, Theorem 7.2 shows that $\delta_b = a' - (s - b)$ exactly if no double rounding slip occurred in Line 2: in such a case, $\delta_a + \delta_b = (a + b - s)$, which implies $t = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a+b-s)) = \mathrm{RN}_p(a + b - s)$ (using Remark 7.8).

Thus, in the following, we assume that a double rounding slip occurred in Line 2, i.e., when computing $s - b$. Notice that this implies from Remark 7.6 that a' is even. Notice that Equation (7.4) in Remark 7.3 implies that $e_b \leqslant e'_a - p' - 1$.

Also, we know that

- $\delta_a = a - a'$ and $b' = s - a'$ exactly;
- all variables $(s, a', b', \delta_a, \delta_b, t)$ are multiples of $2^{e_b - p + 1}$.

Since a double rounding slip occurred in Line 2, we have

$$s - b = a' + i \cdot 2^{e'_a - p} + j \cdot \epsilon,$$

where $i = \pm 1$ (or $\pm\frac{1}{2}$ in the case a' is a power of 2), $j = \pm 1$ and $0 \leqslant \epsilon \leqslant 2^{e'_a - p - p'}$. Since $b' = s - a'$ exactly, we deduce

$$b' = b + i \cdot 2^{e'_a - p} + j \cdot \epsilon,$$

hence,

$$b - b' = -i \cdot 2^{e'_a - p} - j \cdot \epsilon,$$

so that, since it is a multiple of $2^{e_b - p + 1}$, $b - b'$ fits in at most $(e'_a - p) - (e_b - p + 1) + 1 = e'_a - e_b \leqslant e_a - e_b + 1$ bits. Hence, if $e_a - e_b \leqslant p - 1$ then $b - b'$ is a precision-p floating-point number, therefore $\delta_b = b - b'$ exactly. It follows that $\delta_a + \delta_b = a - a' + b - b' = a - a' + b - (s - a') = a + b - s$, so that $t = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a + b - s)) = \mathrm{RN}_p(a + b - s)$ (using Remark 7.8).

Furthermore, if $e_a - e_b \geqslant p + 2$, then one easily checks that $s = a' = a$, $b' = \delta_a = 0$, and $t = \delta_b = b$, which is the desired result.

Hence the last case that remains to be checked is the case $e_a - e_b \in \{p, p + 1\}$. Notice that in that case, if a is not a power of 2, s is necessarily equal to a^-, a, or a^+, where a^- and a^+ are the floating-point predecessor and successor of a. If a is a power of 2, s can also be equal to a^{--} (when $a > 0$) or a^{++} (when $a < 0$). To simplify the presentation, we now assume $a > 0$ (otherwise, it suffices to change the signs of a and b).

1. **if a is not a power of 2,** then s is equal to a^-, a, or a^+. Notice that $s = a^- \Rightarrow a' \geqslant s$ (because in that case, $b < 0$), and $s = a^+ \Rightarrow a' \leqslant s$.

   - if $|b|$ is not of the form $\pm 2^{e_a - p} + \epsilon$ with $|\epsilon| \leqslant 2^{e_a - p - p'}$, then there are no double rounding slips in lines (1) and (2) of the algorithm;

- otherwise, if a is even, then (due to the round to nearest *even* rounding rule) $s = a$, and $a' = s = a$, therefore Remark 7.9 implies $t = \mathrm{RN}_p(a+b-s)$;
- otherwise, if a is odd, then (still due to the round to nearest *even* rounding rule) $s = a^+$ or a^- and $a' = s$, so that $b' = 0$, which implies $\delta_b = b$ and $t = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a-a'+b)) = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a+b-s)) = \mathrm{RN}_p(a+b-s)$.

2. **if a is a power of 2,** i.e., $a = 2^{e_a}$. Notice again that $s < a \Rightarrow a' \geqslant s$ (because in that case, $b < 0$), and $s > a \Rightarrow a' \leqslant s$.

   - if $b \geqslant 0$, then s is equal to a, or a^+.
     - If $s = a^+$ then if $a' = a^+$ then there is no double rounding slip in Line 2 of the algorithm since a' is odd (using Remark 7.6), and if $a' = a$ then Remark 7.9 implies $t = \mathrm{RN}_p(a+b-s)$;
     - now, if $s = a$ then if $a' = a$ then Remark 7.9 implies $t = \mathrm{RN}_p(a+b-s)$, and if $a' = a^-$ then (since a' is odd) there is no double rounding slip in Line 2 of the algorithm (using Remark 7.6).
   - if $b < 0$, then s is equal to a, a^-, or a^{--}.
     - if $s = a$ then $b \geqslant -2^{e_a-p-1} - 2^{e_a-p-p'-1}$. In such a case, there is no double rounding slip when computing $s-b$, i.e., in Line 2 of the algorithm;
     - if $s = a^-$ then if $a' = a^-$ then there is no double rounding slip in Line 2 since a^- is odd, and if $a' = a$ then Remark 7.9 implies $t = \mathrm{RN}_p(a+b-s)$;
     - if $s = a^{--}$ then if $a' = a^-$ then there is no double rounding slip in Line 2 since a^- is odd; if $a' = a$ then Remark 7.9 implies $t = \mathrm{RN}_p(a+b-s)$; and $a' = a^{--}$ is impossible ($s = a^{--}$ implies $-b \geqslant 3 \cdot 2^{e_a-p-1} - 2^{e_a-p-p'-1}$, from which we deduce $s-b > a^-$, which implies $a' \geqslant a^-$).

We therefore deduce

Theorem 7.3. *Assume a radix-2 target floating-point format of precision $p \geqslant 4$, assume that a format of precision $p+p'$, with $p' \geqslant 2$, is available. If a and b are precision-p numbers, and if no overflow occurs, then Algorithm 7.3 satisfies the following property:*

- *if no double rounding slip occurred when computing s (in other words, if $s = \mathrm{RN}_p(a+b)$), then $t = (a+b-s)$ exactly;*
- *otherwise, $t = \mathrm{RN}_p(a+b-s)$.*

Notice that an immediate consequence of Theorems 7.2 and 7.3 is

Corollary 7.1. *The values s and t returned by Algorithms 7.2 and 7.3 satisfy*

$$(s+t) = (a+b)(1+\eta),$$

with $|\eta| \leqslant 2^{-2p} + 2^{-2p-p'} + 2^{-3p-p'}$.

It may be of interest to notice that, even when a double rounding slip occurred when computing s in Fast2Sum or TwoSum, $(a+b)-s$ will very often be exactly representable. More exactly,

Remark 7.10. Assume $p' \geqslant 2$, let a and b be precision-p FP numbers, with $e_a \geqslant e_b$, such that

$$s = \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a+b)\right) \neq \mathrm{RN}_p(a+b),$$

if $a+b-s$ is not a precision-p FP number, then $e_b = e_a - p - 1$.

Proof. Assume that $a+b-s$ is not a precision-p FP number. Without l.o.g., we assume $a+b \geqslant 0$. First, Remark 7.3 implies

$$e_a - p - 1 \leqslant e_b \leqslant e_a - p'. \tag{7.10}$$

Also, Remark 7.5 implies that $\mathrm{RN}_{p+p'}(a+b)$ has the form $g + \frac{1}{2}\,\mathrm{ulp}_p(g)$, where g is a precision-p FP number, which means that

$$a + b = g + \frac{1}{2}\,\mathrm{ulp}_p(g) + \epsilon,$$

with

$$|\epsilon| \leqslant \frac{1}{2}\,\mathrm{ulp}_{p+p'}(g) \text{ and } \epsilon \neq 0.$$

Therefore, we have

$$s = \begin{cases} g & \text{if } g \text{ is even} \\ g^+ = g + \mathrm{ulp}(g) & \text{otherwise.} \end{cases}$$

and

$$\mathrm{RN}_p(a+b) = \begin{cases} g & \text{if } \epsilon < 0 \\ g^+ & \text{if } \epsilon > 0. \end{cases}$$

Hence, s and $\mathrm{RN}_p(a+b)$ differ in two cases:

1. if g is even and $\epsilon > 0$, in which case
$$a + b - s = \frac{1}{2}\,\mathrm{ulp}_p(g) + \epsilon;$$
2. if g is odd and $\epsilon < 0$, in which case
$$a + b - s = -\frac{1}{2}\,\mathrm{ulp}_p(g) + \epsilon.$$

Now, notice that ϵ is an integer multiple of $\mathrm{ulp}(b)$. Equation (7.10) implies that e_g is $e_a - 1$, e_a, or $e_a + 1$. Furthermore,

- If $e_g = e_a - 1$, then $1/2\,\mathrm{ulp}(e_g) = 2^{e_a - p - 1} \leqslant 2^{e_b}$, so that $\pm 1/2\,\mathrm{ulp}(e_g) + \epsilon$ is representable in precision-p FP arithmetic;

- If $e_g = e_a$, then the leftmost bit of the binary representation of $\pm\frac{1}{2}\operatorname{ulp}_p(g)+\epsilon$ is of weight $\leqslant 2^{e_g-p}$, whereas its rightmost nonzero bit has weight $\geqslant 2^{e_b-p+1}$. Hence, if $\pm\frac{1}{2}\operatorname{ulp}_p(g)+\epsilon$ is not a precision-p FP number then

$$e_b - p + 1 < (e_g - p) - p + 1,$$

which implies

$$e_b \leqslant e_a - p - 1.$$

Combined with (7.10), this gives

$$e_b = e_a - p - 1.$$

- If $e_g = e_a + 1$, reasoning as previously, we find that if $\pm\frac{1}{2}\operatorname{ulp}_p(g)+\epsilon$ is not a precision-p FP number then

$$e_b - p + 1 < (e_g - p) - p + 1,$$

which implies

$$e_b \leqslant e_a - p.$$

However, if $e_b \leqslant e_a - p$, then $|b| < \operatorname{ulp}_p(a) = 2^{e_a-p+1}$, therefore (since $|a| \leqslant (2^p - 1)\cdot 2^{e_a-p+1}$), $|a+b| < 2^{e_a+1}$, which implies

$$\mathrm{RN}_{p+p'}(a+b) \leqslant 2^{e_a+1}.$$

Hence, $\mathrm{RN}_{p+p'}(a+b)$ cannot be of the form $g+\frac{1}{2}\operatorname{ulp}(g)+\epsilon$, with $e_g = e_a+1$ and $|\epsilon| \leqslant \frac{1}{2}\operatorname{ulp}_{p+p'}(g)$. □

A consequence of Remark 7.10 will be of interest when discussing the Rump, Ogita and Oishi Splitting algorithm (useful for designing a summation algorithm):

Remark 7.11. If a is a power of 2 (say, $a = 2^{e_a}$) and $|b| \leqslant a$, then the values s and t returned by Algorithm 7.2 or Algorithm 7.3 satisfy

$$t = a + b - s$$

exactly.

Proof. Suppose we have $t \neq a + b - s$. We know that this cannot happen if $s = \mathrm{RN}_p(a+b)$, so we necessarily have

$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a+b)\right) \neq \mathrm{RN}_p(a+b),$$

and $a+b-s$ is not a FP number. Remark 7.10 implies $e_b = e_a - p - 1$, so that

$$|b| < \frac{1}{2}\operatorname{ulp}(a).$$

- if $b > 0$ then the only case for which we may have a double rounding slip (according to Remark 7.5, and the fact that $a + b < a + \frac{1}{2}\operatorname{ulp}_p(a) \Rightarrow \mathrm{RN}_{p+p'}(a+b) \leqslant a + \frac{1}{2}\operatorname{ulp}(a)$) is

$$\mathrm{RN}_{p+p'}(a+b) = 2^{e_a} + \frac{1}{2}\operatorname{ulp}(a),$$

- if $b < 0$ then then the only case for which we may have a double rounding slip (still according to Remark 7.5, and the fact that $a+b > a-\frac{1}{2}\,\mathrm{ulp}(a) = a^-$, so that $\mathrm{RN}_{p+p'}(a+b) \geqslant a^-$) is

$$\mathrm{RN}_{p+p'}(a+b) = 2^{e_a} - \frac{1}{4}\,\mathrm{ulp}(a).$$

In both cases, the round-to-nearest-*even* rounding rule implies $s = 2^{e_a} = a$, so that $a + b - s = b$, which contradicts the assumption that $a + b - s$ is not a FP number. This proof is illustrated by Figure 7.1. □

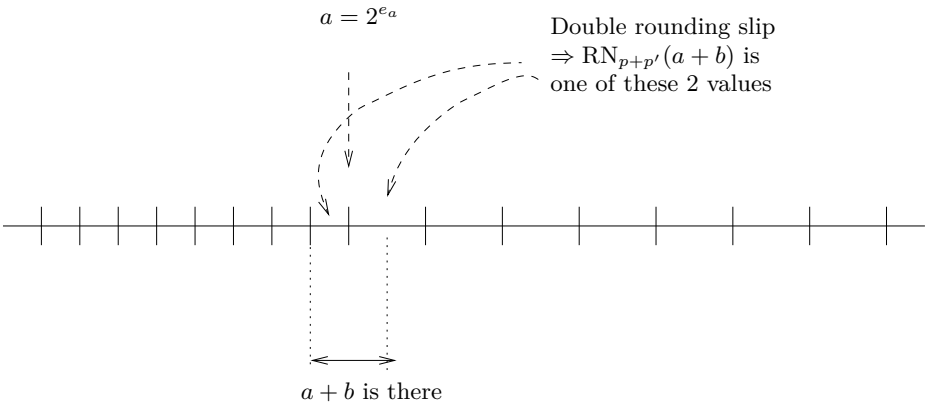

Figure 7.1 – *This figure illustrates the fact that when a is a power of 2, we have $t = a + b - s$ exactly. Here, if a double rounding slip occurs, $\mathrm{RN}_{p+p'}(a+b)$ can take two possible values only, and for each of them $s = a$.*

7.2.4 Consequences of Theorems 7.2 and 7.3 on Summation Algorithms

Many numerical problems require the computation of sums of lots of FP numbers. Several *compensated summation* algorithms use, either implicitly or explicitly, the Fast2Sum or TwoSum algorithms [91, 144, 145, 138, 148]. As a consequence, when double roundings may occur, it is of importance to know if the fact that we can only guarantee that we return the FP number nearest the error of a FP addition (instead of that error itself) may have an influence on the behavior of these algorithms. There is a huge literature on summation algorithms: the purpose of this section is not to examine all published algorithms, just to give a few examples.

Before analyzing other summation methods, let us see what happens with the naive, "recursive sum" algorithm.

The Recursive Sum Algorithm and Kahan's Compensated Summation Algorithm in the Presence of Double Roundings

Let us consider the naive, recursive-sum algorithm, rewritten with double roundings:

Algorithm 7.4: Naive summation algorithm

$r \leftarrow a_1$
for $i = 2$ **to** n **do**
  $r \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(r + a_i))$
end
return r

A straightforward adaptation of the proof for the error bound of the usual recursive sum algorithm without double roundings gives

Property 7.2. *The final value of the variable r returned by Algorithm 7.4 satisfies*

$$\left| r - \sum_{i=1}^{n} a_i \right| \leqslant \gamma'_{n-1} \sum_{i=1}^{n} |a_i|.$$

Without double roundings, the bound is $\gamma_{n-1} \sum_{i=1}^{n} |a_i|$. See Section 7.1.1 for a definition of notations γ_k and γ'_k.

Kahan's compensated summation algorithm, rewritten with double roundings, is as follows:

Algorithm 7.5: Kahan's compensated summation algorithm

$s \leftarrow a_1$
$c \leftarrow 0$
for $i = 2$ **to** n **do**
  $y \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a_i - c))$
  $t \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(s + y))$
  $c \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(\mathrm{RN}_p(\mathrm{RN}_{p+p'}(t - s)) - y))$
  $s \leftarrow t$
end
return s

Goldberg's proof for this algorithm [62] only uses the ϵ-model, so that adaptation to double roundings is straightforward (it suffices to replace $\mathbf{u}$ by $\mathbf{u}'$ in the ϵ-model), and we will immediately deduce that the final value s provided by Algorithm 7.5 satisfies

$$\left| s - \sum_{i=1}^{n} a_i \right| \leqslant \left(2\,\mathbf{u}' + O(n\,\mathbf{u}'^2)\right) \cdot \sum_{i=1}^{n} |a_i|$$

(see Section 7.1.1 for a definition of notations $\mathbf{u}$ and $\mathbf{u}'$). This makes Kahan's compensated summation algorithm very "robust": double roundings have little influence on the error bound. However, when $\sum_{i=1}^{n} |a_i|$ is very large in front of $|\sum_{i=1}^{n} a_i|$, the relative error of Kahan's compensated summation algorithm becomes large. A solution is to use Priest's *doubly compensated* summation algorithm [145]. For that algorithm, the excellent error bound $2u\,|\sum_{i=1}^{n} |a_i||$ will remain true even in the presence of double roundings (the proof essentially assumes faithfully rounded operations). However, it requires a preliminary sorting of the a_i's by magnitude.

In the following, we investigate the potential influence of double roundings on some sophisticated summation algorithms. For most of these algorithms, the proven error bounds (without double roundings) are of the form

$$\left| \text{computed sum} - \sum_{i=1}^{n} a_i \right| \leqslant u \cdot \left| \sum_{i=1}^{n} a_i \right| + \alpha \cdot \sum_{i=1}^{n} |a_i|.$$

Rump, Ogita, and Oishi exhibit a family of algorithms for which, without double roundings, α has the form $O(n^K 2^{-Kp})$. As we will see, that property will be (roughly) preserved when $K = 2$. However, for the more subtle algorithms—for which $K \geqslant 3$—, double roundings may ruin that property.

Rump, Ogita and Oishi's Cascaded Summation Algorithm in the Presence of Double Roundings

The following algorithm was independently introduced by Pichat [141] and by Neumaier [133]:

Algorithm 7.6: Pichat-Neumaier summation algorithm

```
s ← a_1
e ← 0
for i = 2 to n do
    if |s| ⩾ |a_i| then
        (s, e_i) ← Fast2Sum(s, a_i)
    else
        (s, e_i) ← Fast2Sum(a_i, s)
    end
    e ← RN(e + e_i)
end
return RN(s + e)
```

To avoid tests, the algorithm of Pichat and Neumaier can be rewritten using the TwoSum algorithm. This gives the *cascaded summation* algorithm of Rump, Ogita, and Oishi [138]:

Algorithm 7.7: Rump, Ogita, and Oishi's cascaded summation algorithm

```
s ← a_1
e ← 0
for i = 2 to n do
    (s, e_i) ← TwoSum(s, a_i)
    e ← RN(e + e_i)
end
return RN(s + e)
```

Notice that both algorithms will return the same result. In the following, we therefore focus on Algorithm 7.7 only. More precisely, we will be interested here in analyzing the behavior of that algorithm, with double roundings allowed.

That is, we will consider:

Algorithm 7.8: Algorithm 7.7 with double roundings

$s \leftarrow a_1$
$e \leftarrow 0$
for $i = 2$ **to** n **do**
  $(s, e_i) \leftarrow$ TwoSum-with-double-roundings(s, a_i)
  $e \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(e + e_i))$
end
return $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(s + e))$

Define s_i as the value of variable s after the loop of index i (namely, $s_1 = a_1$, and for $i \geqslant 2$, $s_i =$ TwoSum-with-double-roundings(s_{i-1}, a_i)). One easily finds

$$\begin{cases} s_2 + e_2 &= (a_1 + a_2)(1 + \eta^{(2)}) \\ s_i + e_i &= (s_{i-1} + a_i)(1 + \eta^{(i)}) \\ |e_i| &\leqslant \mathbf{u}'(1+u)|s_i|, \end{cases}$$

with $|\eta^{(i)}| \leqslant 2^{-2p} + 2^{-2p-p'} + 2^{-3p-p'}$ (from Corollary 7.1). Therefore,

$$\begin{aligned}
&s_n + (e_n + e_{n-1} + \cdots + e_2) \\
&= (s_n + e_n) + (e_{n-1} + e_{n-2} + \cdots + e_2) \\
&= (s_{n-1} + a_n)(1 + \eta^{(n)}) + (e_{n-1} + e_{n-2} + \cdots + e_2) \\
&= a_n(1 + \eta^{(n)}) + s_{n-1}\eta^{(n)} + (s_{n-1} + e_{n-1}) + (e_{n-2} + \cdots + e_2) \\
&= a_n(1 + \eta^{(n)}) + s_{n-1}\eta^{(n)} + (s_{n-2} + a_{n-1})(1 + \eta^{(n-1)}) + (e_{n-2} + \cdots + e_2) \\
&= \cdots \\
&= a_n(1 + \eta^{(n)}) + a_{n-1}(1 + \eta^{(n-1)}) + \cdots + a_3(1 + \eta^{(3)}) \\
&+ s_{n-1}\eta^{(n)} + s_{n-2}\eta^{(n-1)} + \cdots + s_2\eta^{(3)} \\
&+ (s_2 + e_2) \\
&= \sum_{i=3}^{n} a_i(1 + \eta^{(i)}) + \sum_{i=3}^{(n)} s_{i-1}\eta^{(i)} + (a_1 + a_2)(1 + \eta^{(2)}),
\end{aligned}$$

From which we deduce

$$s_n + \sum_{i=2}^{n} e_i = \sum_{i=1}^{n} a_i + \eta \cdot \sum_{i=1}^{n} |a_i| + \eta' \sum_{i=2}^{n} |s_i|, \tag{7.11}$$

with $|\eta|, |\eta'| \leqslant 2^{-2p} + 2^{-2p-p'} + 2^{-3p-p'}$.

Let $|E|$ be obtained by computing $\sum_{i=2}^{n} e_i$ by the recursive summation algorithm (possibly with double roundings), from Property 7.2, we have

$$\left| E - \sum_{i=2}^{n} e_i \right| \leqslant \gamma'_k \cdot \sum_{i=2}^{n} |e_i|. \tag{7.12}$$

Let us now bound $\sum_{i=2}^{n} |e_i|$. We already have

$$|e_i| \leqslant \left(2^{-p} + 2^{-p-p'} + 2^{-2p} + 2^{-2p-p'}\right) \cdot |s_i|.$$

Now,

$$|s_2| \leqslant (|a_1| + |a_2|) \cdot (1 + \gamma_1'),$$

so that

$$\begin{array}{rcl} |s_3| & \leqslant & [(|a_1| + |a_2|)(1 + \gamma_1') + |a_3|] \cdot (1 + \gamma_1') \\ & < & (|a_1| + |a_2| + |a_3|) \cdot (1 + \gamma_2'), \end{array}$$

and, by induction

$$\begin{array}{rcl} |s_j| & < & (|a_1| + |a_2| + \cdots + |a_j|) \cdot (1 + \gamma_{j-1}') \\ & < & (|a_1| + |a_2| + \cdots + |a_n|) \cdot (1 + \gamma_{n-1}'). \end{array}$$

Therefore,

$$\sum_{i=2}^{n} |s_i| \leqslant (n-1) \cdot (1 + \gamma_{n-1}') \cdot \sum_{i=1}^{n} |a_i|, \tag{7.13}$$

which implies

$$\sum_{i=2}^{n} |e_i| \leqslant (n-1)(1 + \gamma_{n-1}') \left(2^{-p} + 2^{-p-p'} + 2^{-2p} + 2^{-2p-p'}\right) \sum_{i=1}^{n} |a_i|. \tag{7.14}$$

Hence,

$$\sum_{i=1}^{n} a_i = s_n + E + \rho, \tag{7.15}$$

where

$$|\rho| < \left(\sum_{i=1}^{n} |a_i|\right) \times \kappa, \tag{7.16}$$

with

$$\begin{array}{rcl} \kappa & = & \eta + (n-1) \cdot (1 + \gamma_{n-1}') \\ & & \cdot \left(\eta' + \left(2^{-p} + 2^{-p-p'} + 2^{-2p} + 2^{-2p-p'}\right) \gamma_{n-2}'\right). \end{array} \tag{7.17}$$

We remind the reader that $\gamma_{n-2}' = (n-2)\,\mathbf{u}'/(1-(n-2)\,\mathbf{u}')$. Assuming $p \geqslant 8$ and $p' \geqslant 4$, we find

$$\left(2^{-p} + 2^{-p-p'} + 2^{-2p} + 2^{-2p-p'}\right) \mathbf{u}' \leqslant 2^{-2p} + 2^{-2p-p'+1} + \frac{2^{-2p}}{100}. \tag{7.18}$$

Let us now assume $|(n-1)\,\mathbf{u}'| < 1/2$, which implies $1/(1-(n-2)) < 2$. From (7.18), we deduce

$$\begin{array}{l} \left(2^{-p} + 2^{-p-p'} + 2^{-2p} + 2^{-2p-p'}\right) \gamma_{n-2}' \\ \leqslant (2n-4) \cdot \left(2^{-2p} + 2^{-2p-p'+1} + \dfrac{2^{-2p}}{100}\right). \end{array} \tag{7.19}$$

Similarly, still assuming $p \geqslant 8$ and $p' \geqslant 4$,

$$\begin{aligned} |\eta'| &\leqslant 2^{-2p} + 2^{-2p-p'} + 2^{-3p-p'} \\ &< 2^{-2p} + 2^{-2p-p'+1}. \end{aligned} \tag{7.20}$$

By combining (7.19) and (7.20), we obtain

$$\begin{aligned} &\left|\eta' + \left(2^{-p} + 2^{-p-p'} + 2^{-2p} + 2^{-2p-p'}\right)\gamma'_{n-2}\right| \\ &< (2n-3)\cdot\left(2^{-2p} + 2^{-2p-p'+1} + \frac{2^{-2p}}{100}\right). \end{aligned}$$

Our assumption $|(n-1)\,\mathbf{u}'| < 1/2$ implies $\gamma'_{n-1} < 1$, therefore the term

$$(n-1)(1+\gamma'_{n-1})$$

in (7.16) is less than $(2n-2)$. From all this, we deduce that the term

$$\left[\eta + (n-1)(1+\gamma'_{n-1})\cdot\left(\eta' + \left(2^{-p} + 2^{-p-p'} + 2^{-2p} + 2^{-2p-p'}\right)\gamma'_{n-2}\right)\right]$$

in (7.16) is less than

$$\left(4n^2 - 10n - 5\right)\cdot\left(2^{-2p} + 2^{-2p-p'+1} + \frac{2^{-2p}}{100}\right). \tag{7.21}$$

Now, from (7.15), the final value, say σ, returned by Algorithm 7.8, satisfies

$$\begin{aligned} \sigma &= (s_n + E)\cdot(1+\theta'), \text{ with } |\theta'| \leqslant \mathbf{u}', \\ &= \left(\sum_{i=1}^{n} a_i - \rho\right)(1+\theta'). \end{aligned}$$

using (7.21), this implies

$$\begin{aligned} \left|\sigma - \sum_{i=1}^{n} a_i\right| &\leqslant \mathbf{u}'\cdot\left|\sum_{i=1}^{n} a_i\right| \\ &+ (1+\mathbf{u}')\left(4n^2 - 10n - 5\right)\left(2^{-2p} + 2^{-2p-p'+1} + \frac{2^{-2p}}{100}\right)\sum_{i=1}^{n}|a_i|. \end{aligned}$$

An elementary calculation, still assuming $p \geqslant 8$ and $p' \geqslant 4$, shows that

$$(1+\mathbf{u}')\cdot\left(2^{-2p} + 2^{-2p-p'+1} + \frac{2^{-2p}}{100}\right) \leqslant 2^{-2p} + 2^{-2p-p'+1} + \frac{3\cdot 2^{-2p}}{200},$$

which gives

Theorem 7.4. *Assuming $p \geqslant 8$, $p' \geqslant 4$, and $n < \dfrac{1}{2\,\mathbf{u}'}$, the final value σ returned by Algorithm 7.8 satisfies*

$$\begin{aligned} \left|\sigma - \sum_{i=1}^{n} a_i\right| &\leqslant \left(2^{-p} + 2^{-p-p'} + 2^{-2p-p'}\right)\cdot\sum_{i=1}^{n} a_i \\ &+ 2^{-2p}\cdot\left(4n^2 - 10n - 5\right)\cdot\left(1 + 2^{-p'+1} + \frac{3}{200}\right)\cdot\sum_{i=1}^{n}|a_i|. \end{aligned}$$

In that case, the final result is not so different from the classical, double-rounding-free, result: in that classical case, the term in front of $\sum_{i=1}^{n} a_i$ is $\mathbf{u} = 2^{-p}$, and the term in front of $\sum_{i=1}^{n} |a_i|$ is γ_{n-1}^2. Hence the Cascaded Summation algorithm is "robust" and can be used safely, even when double roundings may happen: the error bound is slightly larger but remains of the same order of magnitude.

However, more subtle algorithms, that return a more accurate result (assuming no double roundings) when

$$\frac{\sum_{i=1}^{n} |a_i|}{|\sum_{i=1}^{n} a_i|}$$

is very large, may be of less interest when double roundings may happen, unless we have some additional information on the input data that allows one to make sure there will be no problem. Consider for instance the K-fold summation algorithm of Rump, Ogita and Oishi, defined as follows:

Algorithm 7.9: VecSum

Input: $a = (a_1, a_2, \ldots, a_n)$
$p \leftarrow a$
for $i = 2$ **to** n **do**
  $(p_i, p_{i-1}) \leftarrow \mathrm{TwoSum}(p_i, p_{i-1})$
end
return p

Algorithm 7.10: Rump, Ogita and Oishi's K-fold summation algorithm

for $k = 1$ **to** $K - 1$ **do**
  $a \leftarrow \mathrm{VecSum}(a)$
end
$c = a_1$
for $i = 2$ **to** $n - 1$ **do**
  $c \leftarrow \mathrm{RN}(c + a_i)$
end
return $\mathrm{RN}(a_n + c)$

If double roundings are not allowed, Rump, Ogita, and Oishi show that if $4n\,\mathbf{u} < 1$, the final result σ returned by Algorithm 7.10 satisfies

$$\left|\sigma - \sum_{i=1}^{n} a_i\right| \leqslant (\mathbf{u} + \gamma_{n-1}^2)\left|\sum_{i=1}^{n} a_i\right| + \gamma_{2n-2}^{K} \sum_{i=1}^{n} |a_i|. \tag{7.22}$$

If a double-rounding slip occurs in the first call to VecSum, an error as large as $2^{-2p} \max |a_i|$ may be produced. Hence, it will not be possible to show a final error bound better than $2^{-2p} \max |a_i| \geqslant \frac{2^{-2p}}{n} \sum_{i=1}^{n} |a_i|$ when double roundings are allowed. In practice (since double rounding slips are not so frequent, and do not always change the result of TwoSum when they occur), the K-fold summation algorithm will almost always return a result that satisfies a bound close to the one given by (7.22), but exceptions may occur. Consider the following example (with $n = 5$, but easily generalizable to any larger value of n):

$$(a_1, a_2, a_3, a_4, a_5) = \left(2^{p-1} + 1, \frac{1}{2} - 2^{-p-1}, -2^{p-1}, -2, \frac{1}{2}\right)$$

and assume that Algorithm 7.10 is run with double roundings, with $1 \leqslant p' \leqslant p$. One may easily check that in the first addition of the first TwoSum of the first call to VecSum (i.e., when adding a_1 and a_2), a double rounding slip occurs, so that immediately after this first Fast2Sum, $p_2 = 2^{p-1} + 2$ and $p_1 = -1/2$, so that $p_1 + p_2 \neq a_1 + a_2$. At the end of the first call to VecSum, the returned vector is

$$\left(-\frac{1}{2}, 0, 0, 0, \frac{1}{2}\right)$$

so that Algorithm 7.10 will return 0 whatever the value of K, whereas the exact sum of the $a_i's$ is -2^{-p-1}. Hence (since $\sum |a_i| = 2^p + 4 - 2^{-p-1} \approx 2^p$), the final error of Algorithm 7.10 is approximately $2^{-2p-1} \sum |a_i|$, whatever the value of K.

This example shows that if we wish to be sure of getting error bounds of the order of magnitude of the one given by (7.22) when using the K-fold summation algorithm with $K \geqslant 3$, we need to select compilation switches that prevent double roundings from occurring, unless we have additional information on the input data (such as all values having the same order of magnitude) that allow one to use Remark 7.10 to show that Fast2Sum and TwoSum will return an exact result, even in the presence of double roundings.

7.3 Formal Setup in the Coq Proof Assistant

Proofs in computer arithmetic are somewhat complex: they are frequently based on the enumeration of many possible cases, so that one may very easily overlook one of these cases. To avoid this problem and get more confidence in our proofs, we are working on the formal proof of our theorems using the Coq proof assistant [10]. We thus have completed the formal proof of Theorem 7.2 page 163 and of all the support results that we have used to derive its pen-and-paper proof; the formalization[1] is based on the Flocq library [22].

The section is organized as follows:

- In Section 7.3.1, we start by presenting the key concepts of the Flocq library that are required to position our formalization;
- In Section 7.3.2, we describe the theory on midpoints that we developed using the formalism of Flocq, with a special focus on genericity;
- In Section 7.3.3, we summarize the formalization of the various remarks that we stated previously in Section 7.2.1, with the help of our theory on midpoints and a few extra support results;
- Finally Section 7.3.4 is devoted to the formalization of Theorem 7.2 itself.

7.3.1 Technicalities of the Flocq Library

The Flocq library was introduced in Section 3.2.7, but for an understanding of the rest of the chapter, we need to go into the implementation details of Flocq, which is the topic of the present subsection.

[1] The Coq development is available at http://tamadi.gforge.inria.fr/DblRnd/

We will thus highlight some key features of the Flocq library, and describe most of its basic formal definitions, while referring to the presentation given in Chapter 2 if need be.

First, *FP numbers* are defined as pairs $f = (M_f, q_f)$ of signed integers, along with projections $M_f = \texttt{Fnum}(f)$ and $q_f = \texttt{Fexp}(f)$ which will respectively return the *integral significand* and the *quantum exponent* of the FP number f (we reuse the terminology given in Definitions 2.2 and 2.3 from Chapter 2, but no normalization assumption is required in the present context since we only deal with *the components of a FP representation*):

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

Note the presence of the parameter β : `radix` inside the type of this record (`radix` being defined as the type $\mathbb{Z} \cap [2, +\infty[$): this allows one to easily define the *value* of such FP numbers, depending on the radix:

$$\forall \beta : \texttt{radix}, \quad \forall f : \texttt{float}(\beta), \quad \mathrm{F2R}(f) := M_f \times \beta^{q_f} \in \mathbb{R}.$$

Then, the concept of rounding mode is first defined in a generic fashion (without dealing yet with FP formats), as predicates $P : \mathbb{R} \longrightarrow \mathbb{R} \longrightarrow \texttt{Prop}$ that satisfy:

$$\texttt{round\_pred}(P) := \begin{cases} \forall x \in \mathbb{R}, \quad \exists f \in \mathbb{R}, \quad P(x, f) \\ \forall x, y, f, g \in \mathbb{R}, \quad P(x, f) \ \wedge \ P(y, g) \ \wedge \ x \leqslant y \implies f \leqslant g, \end{cases}$$

or in other words, a *rounding predicate* P is a 2-place relation on $\mathbb{R}$ that is left-total and monotonically increasing, implying that it is a functional relation.

Note that in this definition of a rounding predicate, both arguments x and f have type $\mathbb{R}$ (while f could have been specified with type `float` or so, as in the Pff library [21, 20, 19], cf. definition `RoundedModeP` in theory `Fround`). This formalization choice is pervasive in the formalism of Flocq and contributes to increase the user-friendliness of the library.

In particular, relying on the previous definitions of F2R, Flocq defines five standard formats: FIX (fixed-point, i.e., with a constant exponent), FLX (floating-point with unbounded exponents), FLXN (normalized floating-point with unbounded exponents, *which is provably equivalent to FLX*), FLT (floating-point with gradual underflow), and FTZ (floating-point with flush-to-zero, i.e., without subnormal numbers). These formats are defined as parameterized predicates over $\mathbb{R}$ named `FIX_format`, `FLX_format`, etc., following the specification recalled in Table 7.1.

Format	Parameters	x belongs to this format if $\exists f, \ x = \mathrm{F2R}(f) \ \wedge \ \ldots$
FIX	$\beta, q_{\min}$	$q_f = q_{\min}$
FLX	β, p	$\lvert M_f \rvert < \beta^p$
FLXN	β, p	$\beta^{p-1} \leqslant \lvert M_f \rvert < \beta^p \ \vee \ x = 0$
FLT	$\beta, p, q_{\min}$	$q_{\min} \leqslant q \ \wedge \ \lvert M_f \rvert < \beta^p$
FTZ	$\beta, p, q_{\min}$	$\left(q_{\min} \leqslant q \ \wedge \ \beta^{p-1} \leqslant \lvert M_f \rvert < \beta^p\right) \ \vee \ x = 0$

Table 7.1 – *Definition of standard formats in Flocq*

We can see here that Flocq encompasses the formalism of the library Pff, given that the only format that is available in Pff corresponds to the FLT format of Flocq.

Furthermore, Flocq provides a generic framework to define and use arbitrary FP formats, which allows one to obtain simpler, and more general statements of FP theorems. This framework relies on the concept of *generic format,* which is entirely determined by a radix β and a function $\varphi : \mathbb{Z} \longrightarrow \mathbb{Z}$. In order to explain the semantics of such functions φ, we need to mention the following definitions (we reuse the notations from [22]):

Definition 7.1 (Slice). The *slice* of a given nonzero real number x in radix β is the unique integer $n = \mathrm{slice}(x)$ satisfying

$$\beta^{n-1} \leqslant |x| < \beta^n. \tag{7.23}$$

In other words, we have $\mathrm{slice}(x) = \lfloor \log_\beta(x) \rfloor + 1$.

Definition 7.2 (Canonical exponent). The *canonical exponent* of a given nonzero real number x with respect to β and φ is defined by

$$\mathrm{cexp}(x) = \varphi\left(\mathrm{slice}(x)\right) \in \mathbb{Z}. \tag{7.24}$$

Definition 7.3 (Scaled mantissa). The *scaled mantissa* of a given real number x with respect to β and φ is defined by

$$\mathrm{smant}(x) = x \cdot \beta^{-\,\mathrm{cexp}(x)}. \tag{7.25}$$

Notice that mathematically speaking, "slice(0)" and "cexp(0)" are meaningless, while the value that we expect for "smant(0)" is zero. In the Coq formalization, $\mathrm{slice}(\cdot)$ is defined as an opaque, total function along with a proof that characterizes the value of $\mathrm{slice}(x)$ for all nonzero real x. Consequently, "the value slice(0) exists" (since $\mathrm{slice}(\cdot)$ is a *total* function), but no information can be retrieved about this value (thanks to the mechanism of *opacity*). Then $\mathrm{smant}(\cdot)$ is directly defined in Coq using (7.25). Despite the fact that there is no information on the value of cexp(0), we can formally prove that $\mathrm{smant}(0) = 0$, given that $0 \in \mathbb{R}$ is an absorbing element.

Definition 7.4 (Integer part). The *integer part* (rounded towards zero) of a given real number x is defined by

$$\mathcal{Z}(x) = \begin{cases} \lfloor x \rfloor & \text{if } x \geqslant 0 \\ \lceil x \rceil & \text{if } x \leqslant 0. \end{cases} \tag{7.26}$$

Now we can introduce the core definition:

Definition 7.5 (Generic format). The *generic format* associated with β and φ is defined by

$$\mathbb{F}_{\beta,\varphi} := \left\{ x \in \mathbb{R} \,\middle|\, x = \mathcal{Z}(\mathrm{smant}(x)) \cdot \beta^{\mathrm{cexp}(x)} \right\}. \tag{7.27}$$

First, notice that the integer part $\mathcal{Z}$ involved in (7.27) could be replaced just as well with $\lfloor \cdot \rfloor$ or $\lceil \cdot \rceil$, given that we have $\forall x \in \mathbb{R},\ x = \mathrm{smant}(x) \cdot \beta^{\mathrm{cexp}(x)}$ and that the "set of invariant points" is the same for these three functions (it is $\mathbb{Z}$).

Second, we can see from Definition 7.5 that the previously defined integer $\mathrm{cexp}(x)$ corresponds to *the quantum exponent of* x and that $\mathcal{Z}\left(\mathrm{smant}(x)\right)$ corresponds to *the integral significand of* $x \in \mathbb{F}_{\beta,\varphi}$ (cf. Definition 2.3 on page 16).

Then it should be noted that an arbitrary function φ might not relevantly describe what we expect for a FP format. Flocq thus defines a predicate `valid_exp` that formalizes some general constraints on such functions φ.

Definition 7.6 (Valid format). We say that a function $\varphi : \mathbb{Z} \longrightarrow \mathbb{Z}$ defines a *valid generic format* if:

$$\forall k \in \mathbb{Z}, \begin{cases} \varphi(k) < k \implies \varphi(k+1) \leqslant k \\ k \leqslant \varphi(k) \implies \begin{cases} \varphi(\varphi(k)+1) \leqslant \varphi(k) \\ \forall \ell \in \mathbb{Z},\ \ell \leqslant \varphi(k) \implies \varphi(\ell) = \varphi(k). \end{cases} \end{cases} \tag{7.28}$$

The interested reader will be able to find more intuition on these constraints in [22, Section III.D].

Remark 7.12 (Standard formats). Note that all the standard formats mentioned in Table 7.1 can be expressed as instances of Definition 7.5, using the following functions:

$$\begin{aligned} \texttt{FIX\_exp}(q_{\min}) &: k \in \mathbb{Z} \mapsto q_{\min} \\ \texttt{FLX\_exp}(p) &: k \in \mathbb{Z} \mapsto k - p \\ \texttt{FLT\_exp}(p) &: k \in \mathbb{Z} \mapsto \begin{cases} k-p & \text{if } k-p \geqslant q_{\min} \\ q_{\min} & \text{otherwise} \end{cases} \\ \texttt{FTZ\_exp}(p, q_{\min}) &: k \in \mathbb{Z} \mapsto \begin{cases} k-p & \text{if } k-p \geqslant q_{\min} \\ q_{\min}+p-1 & \text{otherwise,} \end{cases} \end{aligned}$$

(where it can be noted that `FTZ_exp` is not a monotone function from $\mathbb{Z}$ to $\mathbb{Z}$). Then, the equivalence between the definitions of Table 7.1 and the generic formalism is given by theorems such as the following:

```
Theorem FLX_format_generic :
  forall (beta : radix) (p : Z), 0 < p -> forall x : R,
    FLX_format beta p x <-> generic_format beta (FLX_exp p) x.
```

In the sequel, we will often deal with the generic formalism, and omit the suffix `_exp` for the standard φ functions: e.g., we will notate $\mathrm{FLX}(p) :=$ `(FLX_exp p)` and $\mathbb{F}_{\beta,\mathrm{FLX}(p)} :=$ `(generic_format beta (FLX_exp p) x)`.

Finally, Flocq defines rounding modes associated with generic formats in the following way:

Definition 7.7 (Generic rounding modes). For any generic format $\mathbb{F}_{\beta,\varphi}$, a *generic rounding mode* is defined as an instance of the function

$$\texttt{round}(\beta, \varphi, \mathrm{Zrnd}) : x \in \mathbb{R} \longmapsto \mathrm{F2R}\Big(\mathrm{Zrnd}(\mathrm{smant}(x)),\ \mathrm{cexp}(x)\Big) \in \mathbb{R},$$

for a given function $\mathrm{Zrnd} : \mathbb{R} \longrightarrow \mathbb{Z}$ satisfying the conditions

$$\begin{cases} \forall n \in \mathbb{Z} \subset \mathbb{R}, \quad \mathrm{Zrnd}(n) = n, \\ \forall x, y \in \mathbb{R}, \quad x \leqslant y \implies \mathrm{Zrnd}(x) \leqslant \mathrm{Zrnd}(y). \end{cases}$$

We can now specialize the function Zrnd involved in Definition 7.7 to obtain the desired roundings modes. For instance:

Example 7.1 (Standard rounding modes). The standard roundings modes described in Definition 2.9 on page 19 can be obtained in the following way:

- `round`$(\beta, \varphi, \lfloor\cdot\rfloor)$ corresponds to RD,
- `round`$(\beta, \varphi, \lceil\cdot\rceil)$ corresponds to RU,
- `round`$(\beta, \varphi, \mathcal{Z})$ corresponds to RZ,
- `round`$(\beta, \varphi,$ `ZnearestE`$)$ corresponds to RNE,
- `round`$(\beta, \varphi,$ `ZnearestA`$)$ corresponds to RNA, and
- `round`$(\beta, \varphi, ($`Znearest choice`$))$ corresponds to an arbitrary rounding-to-nearest mode RN (the variable `choice` having type $\mathbb{Z} \longrightarrow$ `bool`).

7.3.2 Formalization of a Generic Theory on Midpoints

First, for any radix-β and any FP format $\mathbb{F}_{\beta,\varphi}$, we formalize three generic definitions of the set of midpoints $\mathcal{M}_{\beta,\varphi}$ (as well as some equivalent formulations that do not involve any division, which we will not mention here for the sake of brevity):

Definition 7.8 (Midpoint, version 1).

$$\forall x \in \mathbb{R}, \quad x \in \mathcal{M}_{\beta,\varphi} \iff x = \mathrm{RD}(x) + \frac{1}{2}\,\mathrm{ulp}(x) \ \wedge \ x \neq 0. \tag{7.29}$$

Definition 7.9 (Midpoint, version 2).

$$\forall x \in \mathbb{R}, \quad x \in \mathcal{M}_{\beta,\varphi} \iff x = \frac{1}{2}\Big(\mathrm{RD}(x) + \mathrm{RU}(x)\Big) \ \wedge \ x \notin \mathbb{F}_{\beta,\varphi}. \tag{7.30}$$

Definition 7.10 (Midpoint, version 3).

$$\forall x \in \mathbb{R}, \quad x \in \mathcal{M}_{\beta,\varphi} \iff x = \frac{1}{2}\Big(\mathrm{RD}(x) + \mathrm{RU}(x)\Big) \ \wedge \ \mathrm{RD}(x) \neq \mathrm{RU}(x). \tag{7.31}$$

Then we formally prove that all these definitions are equivalent, without assuming any hypothesis: for instance the fact that $\mathbb{F}_{\beta,\varphi}$ is a valid format turns out not to be necessary for deriving these equivalences.

The availability of these multiple, equivalent definitions for the notion of midpoint has been helpful at many occasions in the development. For instance, Definition 7.9 can be used to straightforwardly prove that $\mathcal{M}_{\beta,\varphi} \cap \mathbb{F}_{\beta,\varphi} = \varnothing$ (`midpoint_no_format`), while Definition 7.10 (combined with the fact that $\mathrm{RD}(-x) = -\,\mathrm{RU}(x)$) allows one to easily prove that

Property 7.3 (`midpoint_opp`).

$$\forall x \in \mathbb{R}, \quad x \in \mathcal{M}_{\beta,\varphi} \implies -x \in \mathcal{M}_{\beta,\varphi}. \tag{7.32}$$

A key formalized result that may be considered as belonging to "common arithmetic folklore" is given by the following theorem, whose formal verification was actually somewhat tricky due to the large number of new support results and cases analyses it led to:

Theorem 7.5 (`no_midpoint_same_round`). *For any radix β, FP format $\mathbb{F}_{\beta,\varphi}$ and round-to-nearest mode* RN *on $\mathbb{F}_{\beta,\varphi}$, we have*

$$\forall x, y \in \mathbb{R}, \quad x \leqslant y \ \wedge \ \Big(\forall z \in \mathbb{R}, \quad x \leqslant z \leqslant y \implies z \notin \mathcal{M}_{\beta,\varphi}\Big) \implies \mathrm{RN}(x) = \mathrm{RN}(y). \quad (7.33)$$

Theorem 7.5 can be proved as a corollary of the following theorem, which is expressed in a more constructive fashion (specifically, there is an existential quantifier):

Theorem 7.6 (`RN_lt_exists_midpoint`). *For any radix β, FP format $\mathbb{F}_{\beta,\varphi}$ and round-to-nearest mode* RN *on $\mathbb{F}_{\beta,\varphi}$, we have*

$$\forall x, y \in \mathbb{R}, \quad \mathrm{RN}(x) < \mathrm{RN}(y) \implies \exists \mu \in \mathcal{M}_{\beta,\varphi}, \quad x \leqslant \mu \leqslant y. \quad (7.34)$$

The rest of this section will thus be devoted to the formal proof of this Theorem 7.6, whose elaboration led us to formalize a number of results related to midpoints that will be reused in the sequel.

First, we define a generic predicate that will be useful to state and prove results on the relative location of FP numbers and midpoints:

Definition 7.11 (`surround` predicate). We define `surround` as a predicate taking four arguments $P : \mathbb{R} \longrightarrow$ `Prop`, $x \in \mathbb{R}$, $a \in \mathbb{R}$ and $b \in \mathbb{R}$, asserting that a and b are the closest reals surrounding x and satisfying the given predicate P:

$$\forall P, x, a, b, \ \texttt{surround}(P, x, a, b) \iff P(a) \ \wedge \ P(b) \ \wedge \ a \leqslant x < b \ \wedge \ \Big(\forall c \in \mathbb{R}, \ a < c < b \Rightarrow \neg P(c)\Big). \quad (7.35)$$

The fact we require an *asymmetric* double-inequality $a \leqslant x < b$ in Definition 7.11 (instead of $a \leqslant x \leqslant b$) enables us to prove the values a and b at stake are unique, whether or not $P(x)$ is satisfied.

Moreover, we will typically replace the predicate $P : \mathbb{R} \longrightarrow$ `Prop` with either the set $\mathbb{F}_{\beta,\varphi}$, or the set $\mathcal{M}_{\beta,\varphi}$, but other specializations would be possible thanks to the genericity of the predicate `surround`.

In particular, we derive the following theorem that deals with surrounding midpoints, and which constitutes a full generalization of the results given in the second part of Theorem 7.1:

Theorem 7.7 (`surroundP`). *For any radix β and FP format $\mathbb{F}_{\beta,\varphi}$, and for all $x \in \mathbb{F}_{\beta,\varphi}$ such that $x \neq 0$, we have:*

- *$|x|$ is not a power of β, then the midpoints surrounding x are $\mu_1 = x - \frac{1}{2}\,\mathrm{ulp}(x)$ and $\mu_2 = x + \frac{1}{2}\,\mathrm{ulp}(x)$;*
- *if $x > 0$ is a power of β, then the midpoints surrounding x are $\mu_1 = x - \frac{1}{2}\,\mathrm{ulp}(x/\beta)$ and $\mu_2 = x + \frac{1}{2}\,\mathrm{ulp}(x)$;*

- *if $x < 0$ is (the opposite of) a power of β, then the midpoints surrounding x are $\mu_1 = x - \frac{1}{2}\,\mathrm{ulp}(x)$ and $\mu_2 = x + \frac{1}{2}\,\mathrm{ulp}(x/\beta)$.*

Note the location of the division by β: it is $\mathrm{ulp}(x/\beta)$, not $\mathrm{ulp}(x)/\beta$. This allows one to seamlessly handle the case where x/β is a "subnormal power of β," taking advantage of the behavior of the function ulp provided in the Flocq library.

To shorten the statement of the previous theorem, we define the following predicate

Definition 7.12 (Signed powers of β).

$$\forall x \in \mathbb{R}, \quad \texttt{is\_pow}(x) \iff \exists e \in \mathbb{Z}, \quad |x| = \beta^e. \tag{7.36}$$

along with a decidability lemma

```
Lemma is_pow_dec : forall x : R, {is_pow x} + {~ is_pow x}.
```

that we proved *using the total order on* $\mathbb{R}$, and which will allow for building terms by case analysis on `is_pow`.

A useful result related to `is_pow` is given by the following lemma, which is easy to derive from the definition of the predecessor function `pred`:

Property 7.4 (`POSpow_pred`). *For all generic format $\mathbb{F}_{\beta,\varphi}$, we have*

$$\forall x \in \mathbb{F}_{\beta,\varphi}, \quad x > 0 \ \wedge \ \texttt{is\_pow}_\beta(x) \implies \texttt{pred}(x) = x - \mathrm{ulp}(x/\beta). \tag{7.37}$$

Now we focus on the proof of a previously-mentioned key theorem:

Proof of Theorem 7.7. Let $x \in \mathbb{F}_{\beta,\varphi}$ a nonzero FP number. We give below a detailed sketch of the proof of Theorem 7.7 in the case where x is a signed power of β (the case where $|x|$ is not a power of β being very similar, if not simpler).

We discuss on the sign of x:

- Case 1: $x > 0$ is a power of β
  We pose $\mu_1 = x - \frac{1}{2}\,\mathrm{ulp}(x/\beta)$ and $\mu_2 = x + \frac{1}{2}\,\mathrm{ulp}(x)$. We have to prove that μ_1 and μ_2 are the surrounding midpoints of x, that is

$$\mu_1 \in \mathcal{M}_{\beta,\varphi} \ \wedge \ \mu_2 \in \mathcal{M}_{\beta,\varphi} \ \wedge \ \mu_1 \leqslant x < \mu_2 \\ \wedge \ \Big(\forall c \in \mathbb{R}, \ \mu_1 < c < \mu_2 \Rightarrow c \notin \mathcal{M}_{\beta,\varphi}\Big). \tag{7.38}$$

  This can be split into four goals:

  – Goal 1.1: $\mu_1 \in \mathcal{M}_{\beta,\varphi}$?

    To start with, we invoke the following result (`pred_ge_0`) from the Flocq library:

$$\forall x \in \mathbb{R}, \quad 0 < x \ \wedge \ x \in \mathbb{F}_{\beta,\varphi} \implies 0 \leqslant \texttt{pred}(x), \tag{7.39}$$

    then we use Property 7.4 and the definition of μ_1: this implies that we have $0 \leqslant \texttt{pred}(x) < \mu_1 < x$, which allows us to show that

$$\mathrm{RD}(\mu_1) = \texttt{pred}(x) \tag{7.40}$$

and

$$\mathrm{RU}(\mu_1) = x, \tag{7.41}$$

relying on a set of lemmas related to `pred`, RD and RU that generalize some results of the Flocq library. Now we can show that μ_1 satisfies the Definition 7.10 of a midpoint: we have trivially $\mathrm{RD}(\mu_1) \neq \mathrm{RU}(\mu_1)$, and

$$\begin{aligned}\frac{1}{2}\Big(\mathrm{RD}(\mu_1) + \mathrm{RU}(\mu_1)\Big) &= \frac{1}{2}\Big(\texttt{pred}(x) + x\Big)\\ &= \frac{1}{2}\Big(x - \mathrm{ulp}(x/\beta) + x\Big) \quad \text{by Property 7.4}\\ &= x - \frac{1}{2}\,\mathrm{ulp}(x/\beta)\\ &= \mu_1.\end{aligned}$$

– Goal 1.2: $\mu_2 \in \mathcal{M}_{\beta,\varphi}$?
We have $x < \mu_2 < x + \mathrm{ulp}(x)$, and we can directly use lemmas `round_DN_succ` and `round_UP_succ` from the Flocq library to obtain:

$$\mathrm{RD}(\mu_2) = x \tag{7.42}$$

and

$$\mathrm{RU}(\mu_2) = x + \mathrm{ulp}(x). \tag{7.43}$$

Hence we deduce that μ_2 satisfies Definition 7.10: we have $\mathrm{RD}(\mu_2) \neq \mathrm{RU}(\mu_2)$ and

$$\begin{aligned}\frac{1}{2}\Big(\mathrm{RD}(\mu_2) + \mathrm{RU}(\mu_2)\Big) &= \frac{1}{2}\Big(x + x + \mathrm{ulp}(x)\Big)\\ &= x + \frac{1}{2}\,\mathrm{ulp}(x)\\ &= \mu_2.\end{aligned}$$

– Goal 1.3: $\mu_1 \leqslant x < \mu_2$?
It amounts to showing that

$$x - \frac{1}{2}\,\mathrm{ulp}(x/\beta) \leqslant x < x + \frac{1}{2}\,\mathrm{ulp}(x),$$

which is trivial given that $\mathrm{ulp}(y) > 0$ for all $y \neq 0$.

– Goal 1.4: $\forall c \in \mathbb{R},\ \mu_1 < c < \mu_2 \Rightarrow c \notin \mathcal{M}_{\beta,\varphi}$?
Suppose one $c \in \mathbb{R}$ satisfies $\mu_1 < c < \mu_2$ and $c \in \mathcal{M}_{\beta,\varphi}$, and let us try to derive a contradiction. Expanding the Definition 7.10 of a midpoint, we have the following hypotheses:

$$\mu_1 < c < \mu_2, \tag{7.44}$$

$$c = \frac{1}{2}\Big(\mathrm{RD}(c) + \mathrm{RU}(c)\Big), \tag{7.45}$$

$$\mathrm{RD}(c) \neq \mathrm{RU}(c). \tag{7.46}$$

Relying on the total order of $\mathbb{R}$, we have three cases:

* Case 1.4.1: $c < x$
  Using Property 7.4, the definition of μ_1, and (7.44), we have:

$$\texttt{pred}(x) = x - \mathrm{ulp}(x/\beta) < \mu_1 < c < x,$$

  implying

$$\mathrm{RD}(c) = \texttt{pred}(x)$$

  and

$$\mathrm{RU}(c) = x.$$

  Consequently (7.45) becomes

$$c = \frac{1}{2}\Big(\texttt{pred}(x) + x\Big)$$

  hence by Property 7.4,

$$c = x - \frac{1}{2}\,\mathrm{ulp}(x/\beta) = \mu_1,$$

  which contradicts (7.44).
* Case 1.4.2: $c = x$
  This implies $\mathrm{RD}(c) = \mathrm{RD}(x) = x = \mathrm{RU}(x) = \mathrm{RU}(c)$ (given that $x \in \mathbb{F}_{\beta,\varphi}$), which contradicts (7.46).
* Case 1.4.3: $c > x$
  Using (7.44) and the definition of μ_2, we have:

$$x < c < \mu_2 < x + \mathrm{ulp}(x/\beta),$$

  implying

$$\mathrm{RD}(c) = x$$

  and

$$\mathrm{RU}(c) = x + \mathrm{ulp}(x/\beta).$$

  Consequently (7.45) becomes

$$c = x + \frac{1}{2}\,\mathrm{ulp}(x/\beta) = \mu_2,$$

  which contradicts (7.44).

- Case 2: $x < 0$ is (the opposite of) a power of β
  We reuse the result proved in Case 1 for $-x$, which is indeed a positive FP number and a power of β. Consequently we get $\nu_1 = -x - \frac{1}{2}\,\mathrm{ulp}(-x/\beta)$ and $\nu_2 = -x + \frac{1}{2}\,\mathrm{ulp}(-x)$ satisfying

$$\begin{aligned}\nu_1 \in \mathcal{M}_{\beta,\varphi} \;\wedge\; \nu_2 \in \mathcal{M}_{\beta,\varphi} \;\wedge\; \nu_1 \leqslant -x < \nu_2 \\ \wedge\; \Big(\forall c \in \mathbb{R},\; \nu_1 < c < \nu_2 \Rightarrow c \notin \mathcal{M}_{\beta,\varphi}\Big).\end{aligned} \tag{7.47}$$

Taking $\mu_1 := -\nu_2$ and $\mu_2 := -\nu_1$, since $\mathrm{ulp}(-y) = \mathrm{ulp}(y)$ for all $y \neq 0$ and thanks to the Property 7.3, this implies that:

$$\mu_1 \in \mathcal{M}_{\beta,\varphi} \ \wedge \ \mu_2 \in \mathcal{M}_{\beta,\varphi} \ \wedge \ \mu_1 < x \leqslant \mu_2 \\ \wedge \ \Big(\forall c' \in \mathbb{R},\ \mu_1 < c < \mu_2 \Rightarrow c \notin \mathcal{M}_{\beta,\varphi}\Big). \quad (7.48)$$

Thus it only remains to verify that $x < \mu_2 = x + \frac{1}{2}\,\mathrm{ulp}(x/\beta)$, which holds since $\mathrm{ulp}(y) > 0$ for all $y \neq 0$.

This ends the proof of Theorem 7.7. □

It can be noted that among the various inequalities that we used to derive the previous proof, there is no monotonicity assumption on the ulp function, so that our proof is applicable to any valid FP format, including the format FTZ (Flush-To-Zero).

As regards the handling of consecutive FP numbers, the Flocq libraries allows one to use `(fun x => x + ulp beta fexp x)` for the successor of positive FP numbers, and provides a predecessor function `pred` whose properties are only established for positive FP numbers. Thus for greater convenience, we define two (total) functions `Succ` and `Pred` over $\mathbb{R}$ in a symmetric way:

Definition 7.13 (`Pred` and `Succ`). For any FP format $\mathbb{F}_{\beta,\varphi}$ and any real number x, we pose

$$\mathtt{Succ}(x) := \begin{cases} -\mathtt{pred}(-x) & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ x + \mathrm{ulp}(x) & \text{if } x > 0 \end{cases} \ \wedge \ \mathtt{Pred}(x) := \begin{cases} x - \mathrm{ulp}(-x) & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ \mathtt{pred}(x) & \text{if } x > 0. \end{cases} \quad (7.49)$$

Property 7.5 (`Pred` and `Succ`). *The following properties hold:*

$$\begin{aligned} \forall x \in \mathbb{R}, &\quad \mathtt{Pred}(x) \leqslant x \leqslant \mathtt{Succ}(x) \\ \forall x \in \mathbb{R}^*, &\quad \mathtt{Pred}(x) < x < \mathtt{Succ}(x) \\ \forall x \in \mathbb{R}, &\quad \mathtt{Pred}(-x) = -\mathtt{Succ}(x) \end{aligned} \quad (7.50)$$

and for any valid FP format $\mathbb{F}_{\beta,\varphi}$, we have:

$$\begin{aligned} \forall x, y \in \mathbb{F}_{\beta,\varphi}, &\quad x < y \implies \mathtt{Succ}(x) \leqslant y \\ \forall x, y \in \mathbb{F}_{\beta,\varphi}, &\quad x < y \implies x \leqslant \mathtt{Pred}(y) \\ \forall x \in \mathbb{F}_{\beta,\varphi}, &\quad \mathtt{Succ}(x) \in \mathbb{F}_{\beta,\varphi} \\ \forall x \in \mathbb{F}_{\beta,\varphi}, &\quad \mathtt{Pred}(x) \in \mathbb{F}_{\beta,\varphi} \\ \forall x \in \mathbb{R} \setminus \mathbb{F}_{\beta,\varphi}, &\quad \mathrm{RU}(x) \neq 0 \implies \mathtt{Pred}(\mathrm{RU}(x)) = \mathrm{RD}(x)) \\ \forall x \in \mathbb{R}, &\quad \mathrm{RU}(x) \neq 0 \implies \mathtt{Pred}(\mathrm{RU}(x)) \leqslant \mathrm{RD}(x) \\ \forall x \in \mathbb{R} \setminus \mathbb{F}_{\beta,\varphi}, &\quad \mathrm{RD}(x) \neq 0 \implies \mathtt{Succ}(\mathrm{RD}(x)) = \mathrm{RU}(x)) \\ \forall x \in \mathbb{R}, &\quad \mathrm{RD}(x) \neq 0 \implies \mathrm{RU}(x) \leqslant \mathtt{Succ}(\mathrm{RD}(x)) \\ \forall x \in \mathbb{F}_{\beta,\varphi}, \forall y \in \mathbb{R}, &\quad \mathtt{Pred}(x) \leqslant y < x \implies \mathrm{RD}(y) = \mathtt{Pred}(x) \\ \forall x \in \mathbb{F}_{\beta,\varphi}, \forall y \in \mathbb{R}, &\quad x < y \leqslant \mathtt{Succ}(x) \implies \mathrm{RU}(y) = \mathtt{Succ}(x) \end{aligned} \quad (7.51)$$

Most of the results summarized in Property 7.5 admit a simple and direct proof: for instance to prove (7.51) we can combine (7.50) with the following extra result (`lt_UP_le_DN`):

$$\forall x \in \mathbb{R}, \forall y \in \mathbb{F}_{\beta,\varphi}, \quad y < \mathrm{RU}(x) \implies y \leqslant \mathrm{RD}(x).$$

Then we can use this material to prove the following required result:

Property 7.6 (Rounding half-sums). *The following properties hold:*

$$\forall x \in \mathbb{R}, \mathrm{RD}\Big(\frac{1}{2}\Big(\mathrm{RD}(x) + \mathrm{RU}(x)\Big)\Big) = \mathrm{RD}(x)$$
$$\forall x \in \mathbb{R}, \mathrm{RU}\Big(\frac{1}{2}\Big(\mathrm{RD}(x) + \mathrm{RU}(x)\Big)\Big) = \mathrm{RU}(x).$$

whose proof involves three case analyses (sign of $\mathrm{RD}(x)$, whether $x \in \mathbb{F}_{\beta,\varphi}$ or not, and sign of $\mathrm{RU}(x)$).

Then this result allows one to derive the following corollary:

Theorem 7.8 (`midpoint_intro`).

$$\forall x \in \mathbb{R}, \quad x \notin \mathbb{F}_{\beta,\varphi} \implies \frac{1}{2}\Big(\mathrm{RD}(x) + \mathrm{RU}(x)\Big) \in \mathcal{M}_{\beta,\varphi}.$$

which provides another way, in addition to Theorem 7.7, to "introduce midpoints in the context". Both results will be useful to formally prove Theorem 7.6.

Furthermore, a typical situation that will occur in the proof of Theorem 7.6 consists of the case where $x, y \in \mathbb{R}$ satisfy $\mathrm{RN}(x) < \mathrm{RN}(y)$, and $\mathrm{RU}(x) > \mathrm{RD}(y)$. We will call this situation a "crossed configuration", and we show it satisfies the following property:

Property 7.7 (Crossed configuration). *Let x, y be two real numbers such that*

$$\mathrm{RN}(x) < \mathrm{RN}(y) \quad \textit{and} \quad \mathrm{RD}(y) < \mathrm{RU}(x).$$

Then we have

$$\mathrm{RN}(x) = \mathrm{RD}(x) = \mathrm{RD}(y)$$
$$\mathrm{RN}(y) = \mathrm{RU}(y) = \mathrm{RU}(x)$$
$$\mathrm{RU}\Big(\frac{1}{2}\Big(\mathrm{RD}(y) + \mathrm{RU}(x)\Big)\Big) = \mathrm{RU}(x)$$
$$\mathrm{RD}\Big(\frac{1}{2}\Big(\mathrm{RD}(y) + \mathrm{RU}(x)\Big)\Big) = \mathrm{RD}(y).$$

The proof of these results rely on some bookkeeping related to RD and RU, including the following extra result (`float_discr_DN_UP`) that can be viewed as a low-level formulation of the "discreteness" of the set of FP numbers:

$$\forall x, y \in \mathbb{R}, \quad \mathrm{RD}(x) < y < \mathrm{RU}(x) \implies y \notin \mathbb{F}_{\beta,\varphi}.$$

Last but not least, we derive the following result that describe a natural property of rounding-to-nearest, related to the location of a real number with respect to midpoints:

Theorem 7.9 (Strict inequalities and rounding-to-nearest). *For any valid FP format $\mathbb{F}_{\beta,\varphi}$, and for any rounding-to-nearest mode on $\mathbb{F}_{\beta,\varphi}$ denoted by* RN*, we have:*

$$\forall x \in \mathbb{R}, x < \frac{1}{2}\Big(\mathrm{RD}(x) + \mathrm{RU}(x)\Big) \implies \mathrm{RN}(x) = \mathrm{RD}(x)$$
$$\forall x \in \mathbb{R}, x > \frac{1}{2}\Big(\mathrm{RD}(x) + \mathrm{RU}(x)\Big) \implies \mathrm{RN}(x) = \mathrm{RU}(x).$$

The proof directly follows from the properties of RN, RD and RU, using some lemmas of the Flocq library that are expressed in terms of the predicates `Rnd_N_pt`, `Rnd_DN_pt` and `Rnd_UP_pt`.

Now we give a complete pen-and-paper proof of Theorem 7.6:

Proof of Theorem 7.6. Suppose $x, y \in \mathbb{R}$ satisfy $\mathrm{RN}(x) < \mathrm{RN}(y)$, which implies (by monotonicity of RN) that $x < y$. We have to exhibit one midpoint μ satisfying $x \leqslant \mu \leqslant y$. We distinguish between the following two cases (which amounts to saying whether or not there is a floating-point number between x and y):

- Case 1: $\mathrm{RU}(x) \leqslant y$.
  By definition of the rounding towards $-\infty$, this implies $\mathrm{RU}(x) \leqslant \mathrm{RD}(y)$. Either this inequality is strict, or it is an equality:
  - Case 1.1[2]: $\mathrm{RU}(x) < \mathrm{RD}(y)$
    In this part of the proof, we will use Theorem 7.7, which deals with the midpoints surrounding any FP number x *that is nonzero*. We thus start by proving that the following formula holds:
    $$\begin{aligned} 0 < \mathrm{RU}(x) < \mathrm{RD}(y) \ &\vee \ \mathrm{RU}(x) < \mathrm{RD}(y) < 0 \\ &\vee \ \mathrm{RU}(x) \leqslant 0 < \mathrm{RD}(y) \\ &\vee \ \mathrm{RU}(x) < 0 \leqslant \mathrm{RD}(y), \end{aligned}$$
    then we inspect each case:
    - Case 1.1.1: $0 < \mathrm{RU}(x) < \mathrm{RD}(y)$
      We invoke Theorem 7.7 for $t := \mathrm{RU}(x) > 0$, which distinguishes between three cases and provides two surrounding midpoints $\mu_1 < t$ and $\mu_2 > t$ in each of these cases:
      - Case 1.1.1.1: $\mathrm{RU}(x)$ is not a power of β
        We take the midpoint $\mu := \mu_2 = \mathrm{RU}(x) + \frac{1}{2}\,\mathrm{ulp}(\mathrm{RU}(x))$ given by Theorem 7.7. The inequality $x \leqslant \mu$ is trivial, while for $\mu \leqslant y$, it suffices to prove that $\mathrm{RU}(x) + \mathrm{ulp}(\mathrm{RU}(x)) \leqslant \mathrm{RD}(y)$. Given that $\mathrm{RU}(x)$ and $\mathrm{RD}(y)$ are FP numbers satisfying $0 < \mathrm{RU}(x) < \mathrm{RD}(y)$, we can deduce that we indeed have $\mathrm{RU}(x) + \mathrm{ulp}(\mathrm{RU}(x)) \leqslant \mathrm{RD}(y)$, using the appropriate lemma of Flocq.
      - Case 1.1.1.2: $\mathrm{RU}(x)$ is a (positive) power of β
        The only change of context with respect to Case 1.1.1.1 is that the first midpoint μ_1 given by Theorem 7.7 has another value, but this does not matter: we take $\mu = \mu_2$ again and we follow exactly the same reasoning as Case 1.1.1.1.

[2] In the Coq formal development, the proof corresponding to this Case 1.1 has been split into five lemmas entitled `round_lt_exists_midpoint_*`, which helped to factorize the code.

- Case 1.1.1.3: $\mathrm{RU}(x)$ is the opposite of a power of β
  This case cannot occur due to a contradiction related to the sign of $\mathrm{RU}(x)$.

* Case 1.1.2: $\mathrm{RU}(x) < \mathrm{RD}(y) < 0$
  To prove this case we can invoke the result corresponding to Case 1.1.1, after replacing x with $-y$, and y with $-x$, given that we have $\mathrm{RU}(x) = -\mathrm{RD}(-x)$ and $\mathrm{RD}(y) = -\mathrm{RU}(-y)$, which implies $0 < \mathrm{RU}(-y) < \mathrm{RD}(-x)$.
* Case 1.1.3: $\mathrm{RU}(x) \leqslant 0 < \mathrm{RD}(y)$
  Similarly, we use Theorem 7.7 for $t := \mathrm{RD}(y) > 0$ which now leads to the following cases:
  - Case 1.1.3.1: $\mathrm{RD}(y)$ is not a power of β
    We take the midpoint $\mu := \mu_1 = \mathrm{RD}(y) - \frac{1}{2}\,\mathrm{ulp}(\mathrm{RD}(y))$ given by Theorem 7.7. In order to show the inequality $x \leqslant \mu$ holds, we rely on the predecessor function (here denoted by $x \longmapsto x^-$) and we prove that

$$\begin{aligned} x &\leqslant \mathrm{RU}(x) \\ &\leqslant \Big(\mathrm{RD}(y)\Big)^- && \text{since } \mathrm{RU}(x) < \mathrm{RD}(y) \\ &= \mathrm{RD}(y) - \mathrm{ulp}(\mathrm{RD}(y)) && \text{as } \mathrm{RD}(y) \text{ is not a power of } \beta \\ &< \mathrm{RD}(y) - \frac{1}{2}\,\mathrm{ulp}(\mathrm{RD}(y)) \\ &= \mu. \end{aligned}$$

    And we also have $\mu = \mathrm{RD}(y) - \frac{1}{2}\,\mathrm{ulp}(\mathrm{RD}(y)) < \mathrm{RD}(y) \leqslant y$.
  - Case 1.1.3.2: $\mathrm{RD}(y)$ is a (positive) power of β
    We take the midpoint $\mu := \mu_1 = \mathrm{RD}(y) - \frac{1}{2}\,\mathrm{ulp}(\mathrm{RD}(y)/\beta)$ given by Theorem 7.7. The inequality $\mu \leqslant y$ is trivial and the proof of $x \leqslant \mu$ similarly uses the predecessor function, but in a different context:

$$\begin{aligned} x &\leqslant \mathrm{RU}(x) \\ &\leqslant \Big(\mathrm{RD}(y)\Big)^- && \text{since } \mathrm{RU}(x) < \mathrm{RD}(y) \\ &= \mathrm{RD}(y) - \mathrm{ulp}(\mathrm{RD}(y)/\beta) && \text{as } \mathrm{RD}(y) \text{ is a positive power of } \beta \\ &< \mathrm{RD}(y) - \frac{1}{2}\,\mathrm{ulp}(\mathrm{RD}(y)/\beta) \\ &= \mu. \end{aligned}$$

  - Case 1.1.3.3: $\mathrm{RD}(y)$ is the opposite of a power of β
    This case cannot occur due to a contradiction related to the sign of $\mathrm{RD}(y)$.
* Case 1.1.4: $\mathrm{RU}(x) < 0 \leqslant \mathrm{RD}(y)$
  To prove this case we can invoke the result corresponding to Case 1.1.3, after replacing x with $-y$, and y with $-x$, given that we have $\mathrm{RU}(x) = -\mathrm{RD}(-x)$ and $\mathrm{RD}(y) = -\mathrm{RU}(-y)$, which implies $\mathrm{RU}(-y) \leqslant 0 < \mathrm{RD}(-x)$.

– Case 1.2: $\mathrm{RU}(x) = \mathrm{RD}(y)$
We pose $g_x = \frac{1}{2}\Big(\mathrm{RD}(x) + \mathrm{RU}(x)\Big)$ and $g_y = \frac{1}{2}\Big(\mathrm{RD}(y) + \mathrm{RU}(y)\Big)$, which are not necessarily midpoints for the moment. There are four cases, depending on whether or not x, y are FP numbers:
  * Case 1.2.1: $x \in \mathbb{F}_{\beta,\varphi}$ and $y \in \mathbb{F}_{\beta,\varphi}$
  This would implies $x = \mathrm{RU}(x) = \mathrm{RD}(y) = y$, which is excluded.
  * Case 1.2.2: $x \in \mathbb{F}_{\beta,\varphi}$ and $y \notin \mathbb{F}_{\beta,\varphi}$
  By Theorem 7.8, this implies that $\mu := g_y = \frac{1}{2}\Big(\mathrm{RD}(y) + \mathrm{RU}(y)\Big)$ is a midpoint. Moreover it satisfies $x \leqslant \mathrm{RU}(x) = \mathrm{RD}(y) \leqslant g_y$, and $g_y \leqslant y$. Indeed, if we had $y < g_y$, by Theorem 7.9 we would have $\mathrm{RN}(y) = \mathrm{RD}(y)$, hence $\mathrm{RN}(x) < \mathrm{RN}(y)$ would become $x < \mathrm{RD}(y)$, which contradicts $x = \mathrm{RU}(x) = \mathrm{RD}(y)$.
  * Case 1.2.3: $x \notin \mathbb{F}_{\beta,\varphi}$ and $y \in \mathbb{F}_{\beta,\varphi}$
  The proof is similar to the previous one for Case 1.2.2, this time with $\mu := g_x$.
  * Case 1.2.4: $x \notin \mathbb{F}_{\beta,\varphi}$ and $y \notin \mathbb{F}_{\beta,\varphi}$
  We first prove that $x \leqslant g_x \vee g_y \leqslant y$ holds. Indeed, if we had $g_x < x$ and $y < g_y$, by Theorem 7.9, we would have $\mathrm{RN}(x) = \mathrm{RU}(x)$ and $\mathrm{RN}(y) = \mathrm{RD}(y)$, hence $\mathrm{RN}(x) = \mathrm{RN}(y)$, which contradicts $\mathrm{RN}(x) < \mathrm{RN}(y)$. Then we have two cases:
    · Case 1.2.4.1: $x \leqslant g_x$
    By Theorem 7.8, this implies that $\mu := g_x$ is a midpoint, which satisfies $x \leqslant \mu \leqslant \mathrm{RU}(x) = \mathrm{RD}(y) \leqslant y$.
    · Case 1.2.4.2: $g_y \leqslant y$
    Similarly, $\mu := g_y$ is a midpoint, which satisfies $x \leqslant \mathrm{RU}(x) = \mathrm{RD}(y) \leqslant \mu \leqslant y$.

Case 2: $y < \mathrm{RU}(x)$
(In other words, we assume there is no FP number between x and y.) In this case we have $\mathrm{RN}(x) < \mathrm{RN}(y)$ and $\mathrm{RD}(y) < \mathrm{RU}(x)$, which means we are in the "crossed configuration," so by Property 7.7, we have:

$$\begin{cases} \mathrm{RN}(x) = \mathrm{RD}(x) = \mathrm{RD}(y) \\ \mathrm{RN}(y) = \mathrm{RU}(x) = \mathrm{RU}(y). \end{cases} \tag{7.52}$$

We pose $g := \frac{1}{2}\Big(\mathrm{RN}(x) + \mathrm{RN}(y)\Big)$, and we easily prove that g satisfies the Definition 7.10 of a midpoint, relying on from Property 7.7. Last but not least, we show that $x \leqslant g \leqslant y$ holds. If it were not the case, we would have $g < x$ or $y < g$. Hence the two following cases:

– Case 2.1: $g < x$
Using (7.52) and the definition of g, this implies $\frac{1}{2}\Big(\mathrm{RD}(x) + \mathrm{RU}(x)\Big) < x$, so by Theorem 7.9, we have $\mathrm{RN}(x) = \mathrm{RU}(x)$, that is by (7.52), $\mathrm{RN}(x) = \mathrm{RN}(y)$, which contradicts $\mathrm{RN}(x) < \mathrm{RN}(y)$.
– Case 2.2: $g > y$
Similarly, this would imply $\mathrm{RN}(y) = \mathrm{RN}(x)$, which is excluded.

This ends the proof of Theorem 7.6. □

In addition to the material presented up to now, we have been led to prove several extra results related to midpoints that are more format-specific: we will mention some of them in the sequel.

7.3.3 Formalization of the Preliminary Remarks

In this section, we describe the formalization of the results mentioned in Section 7.2.1, with the aim to formally proving Theorem 7.2 with the hypothesis that no underflow nor overflow occurs. Thus we focus on the Flocq format FLX that is parameterized by the precision p and which offers a natural background for this aim.

The formal proofs of these preliminary remarks closely follow the pen-and-paper proofs that we gave in Section 7.2.1, but their formalization required some extra support results that we briefly present here. (As regards Remark 7.7, it was already available in Flocq as `Theorem sterbenz`.)

First, formalizing Remark 7.5 led us to prove the following

Theorem 7.10 (`MID_FLX`). *For any even radix β and precision $p \geqslant 1$, we have:*

$$\forall x \in \mathbb{R}, \quad x \in \mathcal{M}_{\beta,\mathrm{FLX}(p)} \implies x \in \mathbb{F}_{\beta,\mathrm{FLX}(p+1)}, \tag{7.53}$$

whose proof relies on the Definition 7.8 as well as on the following

Property 7.8 (`ulp_FLX_p1`). *For any radix β and $p \in \mathbb{Z}$, we have:*

$$\forall x \in \mathbb{R}, \quad \mathrm{ulp}_{\beta,\mathrm{FLX}(p)}(x) = \mathrm{ulp}_{\beta,\mathrm{FLX}(p+1)}(x) \times \beta. \tag{7.54}$$

Second, proving Remark 7.6 led us to define the following predicate:

Definition 7.14 (`Feven`). We say that a real number x is an *even FP number* with respect to a FP format $\mathbb{F}_{\beta,\varphi}$ if the following condition holds:

$$\texttt{Feven}(\beta,\varphi,x) := \exists f : \texttt{float}(\beta) \text{ such that } \begin{cases} x = \mathrm{F2R}(f), \\ \texttt{canonic}(\beta,\varphi,f) \text{ and} \\ \texttt{Fnum}(f) \in 2\mathbb{Z}, \end{cases} \tag{7.55}$$

where `canonic` is the Flocq predicate saying that a given FP representation is normalized.

So we can derive the following result:

Property 7.9 (`midp_even`). *For any valid FP format $\mathbb{F}_{\beta,\varphi}$ that allows for rounding to nearest even (Flocq condition `Exists_NE`), and denoting by* RNE *the rounding-to-nearest-even mode, we have:*

$$\forall m \in \mathbb{R}, \quad m \in \mathcal{M}_{\beta,\varphi} \implies \texttt{Feven}(\beta,\varphi,\mathrm{RNE}(m)). \tag{7.56}$$

Finally, one of the most technical remark to prove was Remark 7.3, which we split into three Coq lemmas (one for the inequality $e_u - p - 1 \leqslant e_v$, another for the inequality $e_v \leqslant e_u - p'$, and the last one for the inequality $e_w \geqslant e_v + p' + 1$). In particular, the formal proof of the first inequality reuses Properties 7.8 and 7.9 as well as Theorem 7.10, but also the three following extra results:

Property 7.10. *For any valid format $\mathbb{F}_{\beta,\varphi}$ such that φ is monotone (thus excluding FTZ), we have:*

$$\forall x, y \in \mathbb{R}, \quad x \in \mathbb{F}_{\beta,\varphi} \ \wedge \ x \neq 0 \ \wedge \ x - \mathrm{ulp}(x/\beta)/2 < y < x + \mathrm{ulp}(x/\beta)/2 \\ \implies \mathrm{RN}(y) = x, \quad (7.57)$$

which is a corollary of Theorems 7.5 and 7.7 and can be specialized with $x = u$ and $y = u + v$ to show that Inequality (7.5) implies Equality (7.7).

Then, a result related to the predicate `Feven` in radix 2:

Theorem 7.11 (`Feven_pow`). *In radix 2, any* FLX *format with precision $p \geqslant 2$ satisfies:*

$$\forall p \geqslant 2, \quad \forall x \in \mathbb{R}, \quad \texttt{is\_pow}(x) \implies \texttt{Feven}(2, \mathrm{FLX}(p), x). \quad (7.58)$$

(Note that although FLX(1) *is a valid Flocq format, the condition $p \geqslant 1$ would not suffice here.)*

And finally another result that will be combined with Theorem 7.11 to prove Equality (7.6):

Theorem 7.12 (`Feven_DN_UP_incompat`). *For any valid format $\mathbb{F}_{\beta,\varphi}$ such that φ is monotone, we have*

$$\forall x \in \mathbb{R}, \quad x \notin \mathbb{F}_{\beta,\varphi} \ \wedge \ \mathrm{RD}(x) \neq 0 \ \wedge \ \mathrm{RU}(x) \neq 0 \implies \\ \neg\Big(\texttt{Feven}(\beta, \varphi, \mathrm{RD}(x)) \ \wedge \ \texttt{Feven}(\beta, \varphi, \mathrm{RU}(x))\Big). \quad (7.59)$$

Figure 7.2 summarizes the dependencies between the formal proofs of these preliminary remarks as well as of the Theorem 7.2, which is the topic of the next subsection. Note by the way that the node "Hyp. RN = RNE" that appears in the figure stands for the rounding-to-nearest-even rule, which is assumed by two remarks (as well as the other results that rely on them). All other results are formalized in the generic case of a rounding-to-nearest (i.e., with an arbitrary tie-breaking rule "`choice : Z -> bool`").

7.3.4 Formalization of Theorem 7.2 on Fast2Sum

We now focus on the formalization of Theorem 7.2 in the Flocq format FLX.

The corresponding proofs are located in the theory "`Fast2SumWDR`," on top of the other theories gathering the results presented up to now, as shown by Figure 7.3.

First, we can define the context of the proof with the help of the sectioning system of Coq:

```
Section ProofFast2SumWDR.
(* Define the FP formats *)
Variables p p' : Z.
Hypothesis Hp : 2 < p.
Hypothesis Hp' : 1 < p'.
Local Notation fexp := (FLX_exp p).
Local Notation format := (generic_format radix2 fexp).
```

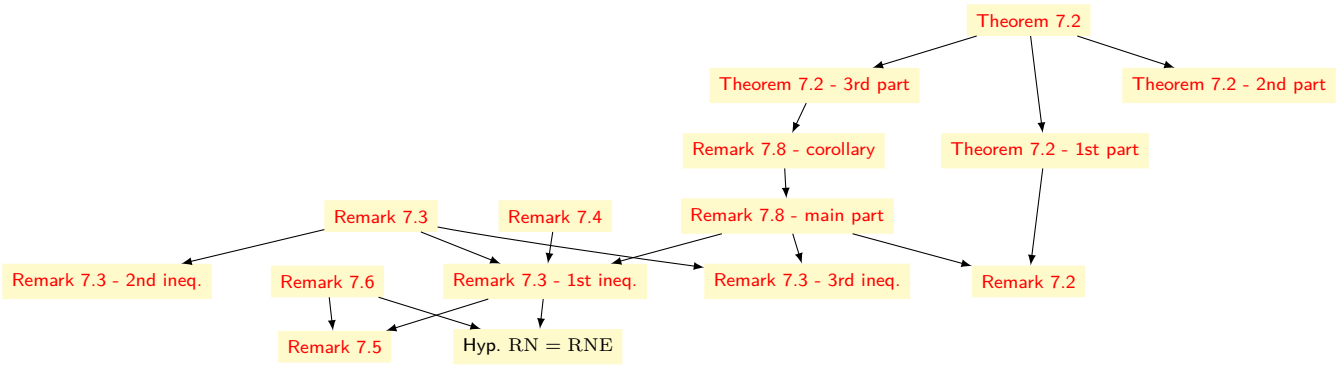

Figure 7.2 – *Dependency graph between formalized Remarks and results on Fast2Sum*

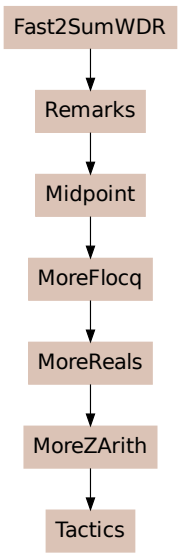

Figure 7.3 – *Dependency graph of the Coq theories on Fast2Sum*

```
Local Notation cexp := (canonic_exp radix2 fexp).
Local Notation mant := (scaled_mantissa radix2 fexp).
Local Notation fexp2 := (FLX_exp (p + p')).
(* Define the rounding modes *)
Variable choice : Z -> bool.
Local Notation rnd_p := (round radix2 fexp (Znearest choice)).
Local Notation rnd_p2 := (round radix2 fexp2 (Znearest choice)).
(* Define the input/output variables of Fast2Sum *)
Variables a b s z t : R.
Hypothesis Fa : format a.
Hypothesis Fb : format b.
Hypothesis sDef : s = rnd_p (rnd_p2 (a + b)).
Hypothesis zDefOr :
  z = rnd_p2 (s - a) \/ z = rnd_p (rnd_p2 (s - a))
                     \/ z = rnd_p (s - a).
Hypothesis tDefOr :
  t = rnd_p (rnd_p2 (b - z)) \/ t = rnd_p (b - z).
Hypothesis cexp_le : cexp b <= cexp a.
```

Now we derive the following technical result, which will allow one to prove Equation (7.9) (by replacing `x` and `m` with s and $\frac{M_a}{2} + \frac{M_b}{2^{\delta+1}} + \epsilon$, respectively):

```
Theorem mant_N : forall x m : R,
  x = (rnd_p (pow (cexp x) * m)) ->
  Ztrunc (mant x) = Znearest choice m.
```

The proof of this result involves several case analyses, notably whether `x` is zero or not and whether it is a power of the radix or not.

Then we closely follow the pen-and-paper proof presented in Section 7.2.2, and use `mant_N` to derive the first part of Theorem 7.2, whose statement in the present context is:

```
Theorem Thm2a : z = s - a.
```

Then the second part of the theorem is as follows:

```
Theorem Thm2b : s = rnd_p (a + b) -> t = a + b - s.
```

whose proof involves the result `plus_error` from the Flocq library, saying that the error of a FP addition is representable in $\mathbb{F}_{\beta,\varphi}$ (provided φ is monotone, which is the case in the present context).

Finally, the proof of the last part of the theorem needs for the rounding mode to be chosen with the round-to-nearest-*even* rule (in order to be able to apply Remark 7.8). We simply state this assumption in the following way:

```
Hypothesis ZNE : choice = fun n => negb (Zeven n).

Theorem Thm2c : s <> rnd_p (a + b) -> t = rnd_p (a + b - s).

End ProofFast2SumWDR.
```

7.4 Conclusion and Perspectives

We have considered the possible influence of double roundings on the algorithms Fast2Sum and TwoSum, which constitute some key building blocks for a number of compensated summation algorithms described in the literature. Although these two algorithms do not behave exactly as when there are no double roundings, they still have interesting properties that can be exploited in a useful way. Depending on the considered applications, these properties may suffice, or specific compilation options should be chosen to prevent double roundings.

We have formally verified in Coq the proof of Theorem 7.2 and the proofs of the Remarks mentioned in Section 7.2.1, relying on the FLX format of the Flocq library, and using the theory on midpoints that we have formalized, which is generic (multi-radix, multi-format) and gathers a number of core results that are widely used in proofs of FP arithmetic. Relying on the material formalized up to now, we now plan to work on the formal proof of other problems related to FP arithmetic (including Theorem 7.3).

Chapter 8

Conclusion and Perspectives

Correct rounding for floating-point implementations of mathematical functions is a key requirement for providing a fully-specified arithmetic, and ensure the portability and the provability of numerical software that is developed upon it. Yet for implementing a given function f with correct rounding at a minimal cost, the *Table Maker's Dilemma* (TMD) has to be solved (offline) for that function. The L and SLZ algorithms were designed to solve this problem, and produce lists of hard-to-round cases. But these calculations are very long, and are performed using heavily optimized programs that implement very complex algorithms: this inevitably casts some doubt on the correctness of their results.

Thus, we have proposed two formal components that can fit in with independent verification and that provide strong guarantees on the results obtained by the SLZ algorithm.

First, in collaboration with members of the TaMaDi project, we have formalized some symbolic-numeric techniques for *Rigorous Polynomial Approximation* (RPA) inside the Coq proof assistant, which allows one to easily compute Taylor approximations of some usual functions in Coq with formally verified error bounds, taking advantage of the computational logic of Coq as well as the CoqInterval library for multiple-precision *interval arithmetic*, on which our library CoqApprox [28] is based.

We plan to investigate further extensions of this library. On the one hand, combining the machinery that we developed in CoqApprox with some Sums-of-Squares techniques may lead to a tool of global optimization, formally verified in Coq. On the other hand, formalizing some other state-of-the-art techniques in symbolic-numeric computation may greatly facilitate the implementation and/or the *a posteriori* verification of RPAs for new functions, directly relying on the ordinary differential equation.

Second, we have designed and formally verified in Coq some *effective checkers of integral-roots certificates* for both univariate and bivariate cases, relying on a uniqueness claim on the modular roots of polynomials over $\mathbb{Z}$ that we have formally proved in Coq, considering the technique of *Hensel lifting* and its generalization to the bivariate case. Then, we have formalized a third effective certificate checker (also in the form of a Boolean function in Coq) that

is specifically designed to check instances of the *Integer Small Value Problem* (ISValP), which is the problem to be dealt with when formally verifying the results of the SLZ algorithm. In particular, this ISValP certificates checker is built upon our bivariate-integral-roots-certificates checker, and likewise for the formal proof of correctness. The timings obtained using our CoqHensel library for typical instances of ISValP are encouraging: in particular, our formalization of "*Integers-Plus-Positive-Exponent*" (IPPE) based on a subset of the CoqInterval library allows one to get a 2x speedup for operations on coefficients.

In a near future, we also plan to implement optimized versions of the operations on polynomials currently implemented in CoqHensel to get further speedups. This should be facilitated by the modular design of the library which properly "encapsulates" the various algorithms and specifications at stake with the help of the Coq `Module` system. Moreover, we would like to investigate the possible extension of our integral-roots-certificates to the case of rational roots, especially for univariate polynomials.

Beyond the development of these components for a formally verified and generic certification chain for the TMD, I have been involved in a joint work for solving the TMD for the bivariate function $(x, y) \mapsto \sqrt{x^2 + y^2}$, for which the IEEE 754-2008 standard recommends correct rounding. We have proposed some "*augmented-precision algorithms*" for computing 2D norms, as well as some tight lower bounds on the nonzero minimum distance from $\sqrt{x^2 + y^2}$ to a midpoint [29]. These results can then be used to provide correct rounding for this function, despite the fact that it admits a large number of exact cases.

Finally, in collaboration with J.-M. Muller and G. Melquiond, we have focused on the *double rounding* phenomenon, which typically occurs in architectures where several precisions are available, when the rounding-to-nearest mode is used, and may lead to what we call a "*double rounding slip.*" In particular, we have studied the potential influence of double roundings on a few usual summation algorithms in radix 2. Then, relying on the Flocq library for multiple-precision floating-point arithmetic in Coq and on the library on *midpoints* that we have developed at this occasion, we have formally verified all the corresponding proofs related to the behavior of the Fast2Sum algorithm in Coq.

The formal verification of these arithmetic results allows one to have a high level of confidence on their proofs, and at the same time leads to a new formalized background that may be reused by further developments. In particular, we plan to work on the formal proof of our results related to the behavior of the TwoSum algorithm with double roundings.

We have carried out these formal developments in Coq using various tools and techniques: from a computational point of view, we thus relied both on autarkic and skeptical approaches, and a part of the proofs were built upon the SSReflect extension of the proof language of Coq. Finally we devoted a particular care to the genericity of our formal developments, either by relying on the Coq `Module` system for separating the specification of computational data structures from their (possibly optimized) implementations, or by introducing general definitions whenever possible, whether implementing RPAs or dealing with midpoints in radix β.

Appendix A

Notations

We summarize below the main mathematical notations used in the manuscript.

- $A \times B$ denotes the Cartesian product of sets A and B, and A^n is a shortcut for $\underbrace{A \times \cdots \times A}_{n \text{ times}}$;
- $A \setminus B$ denotes the subset of A defined by $A \setminus B := \{x \in A \,|\, x \notin B\}$;
- $\# L$ denotes the length of a list L;
- $\mathbb{B}$ denotes the set of booleans, that is $\mathbb{B} := \{\mathsf{true}, \mathsf{false}\}$;
- $\mathbb{P}$ denotes the set of prime numbers;
- $\mathbb{N}$ denotes the set of nonnegative integers;
- $\mathbb{N}^*$ denotes the set of positive integers, that is $\mathbb{N}^* := \mathbb{N} \setminus \{0\}$;
- $\mathbb{Z}$ denotes the ring of signed integers;
- $\mathbb{Q}$ denotes the field of rational numbers;
- $\mathbb{R}$ denotes the field of real numbers;
- $\mathbb{R}^*$ denotes the set of nonzero real numbers;
- $\mathbb{R}_+^*$ denotes the set of positive real numbers;
- $[\![a, b]\!]$ (for $a \leqslant b$ in $\mathbb{Z}$) denotes the set of integers k such that $a \leqslant k \leqslant b$, so $[\![a, b]\!] = \mathbb{Z} \cap [a, b]$;
- $[\![a, b[\![$ (for $a < b$ in $\mathbb{Z}$) denotes the set $\mathbb{Z} \cap [a, b[= \{k \in \mathbb{Z} \,|\, a \leqslant k < b\}$;
- $\lfloor x \rfloor$ (for $x \in \mathbb{R}$) is the largest integer $\leqslant x$; in other words, it is the unique integer $k \in \mathbb{Z}$ such that $k \leqslant x < k + 1$;
- $\lceil x \rceil$ (for $x \in \mathbb{R}$) is the smallest integer $\geqslant x$; in other words, we have $\lceil x \rceil = -\lfloor -x \rfloor$;
- $a \bmod q$ (for $a \in \mathbb{Z}$ and $q \in \mathbb{N}^*$) denotes the remainder of a modulo q, taken in $[\![0, q[\![$;

- $a \equiv b \pmod q$ states the modular equality between a and b; in other words, we have $a \equiv b \pmod q \Leftrightarrow a \bmod q = b \bmod q$;
- $a \operatorname{cmod} q$ denotes the "*centered modulo*" of $a \in \mathbb{Z}$ by $q \in \mathbb{N}^*$, satisfying

$$a \operatorname{cmod} q \equiv a \pmod q \quad \text{and} \quad |a \operatorname{cmod} q| \leqslant \frac{q}{2};$$

  see Definition 2.14 on page 26 for more details on this notion, which also makes sense for any $a \in \mathbb{R}$ and $q \in \mathbb{R}_+^*$ (likewise for the bare modulo);
- $q\mathbb{Z}$ (for $q \in \mathbb{N}$) denotes the principal ideal generated by q, that is the set of integers that are multiples of q;
- $\mathbb{Z}/q\mathbb{Z}$ (for $q \in \mathbb{N}^*$) denotes the ring of integers modulo q, defined as the quotient of $\mathbb{Z}$ with respect to the equivalence relation $\equiv$ defined above, that is $\mathbb{Z}/q\mathbb{Z} = \{q\mathbb{Z}, 1 + q\mathbb{Z}, 2 + q\mathbb{Z}, \ldots, (q-1) + q\mathbb{Z}\}$; we may deal with this set through a set of representatives, like $[\![0, q[\![$;
- $d \mid a$ (for $d, a \in \mathbb{Z}$) denotes the divisibility relation defined by $d \mid a \Leftrightarrow \exists e \in \mathbb{Z},\ a = d \times e$; note that we have $d \mid a \Leftrightarrow a \equiv 0 \pmod d$;
- $\gcd(a, b)$ (for any nonzero pair $(a, b) \in \mathbb{Z}^2$) denotes the greatest common divisor of a and b, taken in $\mathbb{N}$; if $\gcd(a, b) = 1$ we will say that a and b are *coprime*; note also that *if $p \in \mathbb{P}$, an integer a is coprime with (any nontrivial power of) p if and only if $a \not\equiv 0 \pmod p$*;
- a_q^{-1} (for $a \in \mathbb{Z}$ coprime with $q \in \mathbb{N}^*$) denotes the modular inverse of a modulo q, that is the integer $a' \in [\![0, q[\![$ such that $\exists b \in \mathbb{Z},\ aa' + bq = 1$;
- $\mathbb{Z}[X]$ denotes the ring of univariate polynomials with coefficients in $\mathbb{Z}$;
- $\deg(P)$ (assuming $P \in \mathbb{Z}[X]$ is nonzero) denotes the degree of the univariate polynomial $P = \sum_{i=0}^{m} a_i X^i$, that is the largest integer $n \in \mathbb{N}$ such that $a_n \neq 0$;
- P' (for $P \in \mathbb{Z}[X]$) denotes the formal derivative of the univariate polynomial P;
- $\mathbb{Z}[X, Y]$ denotes the ring of bivariate polynomials on $\mathbb{Z}$;
- $\mathcal{M}_{m,n}(A)$ (for any ring A and $m, n \in \mathbb{N}^*$) denotes the set of m-rows, n-columns matrices with coefficients in A, while the notation for order-n square matrices can be shortened to $\mathcal{M}_n(A)$;
- As regards vectors, they will be identified to column matrices; in other words, we assume $A^n \simeq \mathcal{M}_{n,1}(A)$;
- $J_{P_1,P_2}(a, b)$ denotes the Jacobian matrix of the pair (P_1, P_2) of bivariate polynomials on $\mathbb{Z}$ evaluated at $(a, b) \in \mathbb{Z}^2$; in other words,

$$J_{P_1,P_2}(a, b) = \begin{pmatrix} \frac{\partial P_1}{\partial X}(a, b) & \frac{\partial P_1}{\partial Y}(a, b) \\ \frac{\partial P_2}{\partial X}(a, b) & \frac{\partial P_2}{\partial Y}(a, b) \end{pmatrix} \in \mathcal{M}_2(\mathbb{Z}).$$

- $\det(M)$ denotes the determinant of a square matrix M; in particular we have $\begin{vmatrix} a & b \\ c & d \end{vmatrix} := \det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$;
- M_q^{-1} (for $M \in \mathcal{M}_n(\mathbb{Z})$ and $q \in \mathbb{N}^*$) denotes, when it is defined, the modular matrix inverse of M modulo q; for instance we have $\begin{pmatrix} a & b \\ c & d \end{pmatrix}_q^{-1} = \begin{vmatrix} a & b \\ c & d \end{vmatrix}_q^{-1} \cdot \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \bmod q$, where the modulo operation is applied coordinatewise;

We assume this latter convention for the modulo throughout the manuscript, that is: "$A \bmod q$," for $A \in \mathbb{Z}^n$ (resp. for $A \in \mathcal{M}_{m,n}(\mathbb{Z})$) will denote the vector $(A_i \bmod q)_{1 \leqslant i \leqslant n}$ (resp. the matrix $(A_{ij} \bmod q)_{(i,j) \in [\![1,m]\!] \times [\![1,n]\!]}$).

List of Figures

List of Tables

List of Algorithms

Index

D

X

Z

Bibliography

[1] Milton Abramowitz and Irene A. Stegun. *Handbook of mathematical functions with formulas, graphs, and mathematical tables*. Vol. 55. National Bureau of Standards Applied Mathematics Series. For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C., 1964, xiv+1046 pages.

[2] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: *CPP*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011, pp. 135–150. ISBN: 978-3-642-25378-2. DOI: 10.1007/978-3-642-25379-9_12.

[3] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. "Extending Coq with Imperative Features and Its Application to SAT Verification". In: *ITP*. Vol. 6172. LNCS. 2010, pp. 83–98.

[4] I. Babuška. "Numerical stability in mathematical analysis". In: *Proceedings of the 1968 IFIP Congress*. Vol. 1. 1969, pp. 11–23.

[5] Henk Barendregt and Erik Barendsen. "Autarkic Computations in Formal Proofs". In: *J. Autom. Reasoning* 28.3 (2002), pp. 321–336. DOI: 10.1023/A:1015761529444.

[6] Henk Barendregt and Arjeh M. Cohen. "Electronic Communication of Mathematics and the Interaction of Computer Algebra Systems and Proof Assistants". In: *J. Symb. Comput.* 32.1/2 (2001), pp. 3–22. DOI: 10.1006/jsco.2001.0455.

[7] Gilles Barthe, Mark Ruys, and Henk Barendregt. "A Two-Level Approach Towards Lean Proof-Checking". In: *TYPES*. Ed. by Stefano Berardi and Mario Coppo. Vol. 1158. Lecture Notes in Computer Science. Springer, 1995, pp. 16–35. ISBN: 3-540-61780-9. DOI: 10.1007/3-540-61780-9_59.

[8] Alexandre Benoit, Frédéric Chyzak, Alexis Darrasse, Stefan Gerhold, Marc Mezzarobba, and Bruno Salvy. "The Dynamic Dictionary of Mathematical Functions (DDMF)". In: *Mathematical Software - ICMS 2010*. Ed. by Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama. Vol. 6327. Lecture Notes in Computer Science. Springer, 2010, pp. 35–41. ISBN: 978-3-642-15581-9. DOI: 10.1007/978-3-642-15582-6_7.

[9] Daniel J. Bernstein. "Simplified High-Speed High-Distance List Decoding for Alternant Codes". In: *PQCrypto*. Ed. by Bo-Yin Yang. Vol. 7071. Lecture Notes in Computer Science. Springer, 2011, pp. 200–216. ISBN: 978-3-642-25404-8. DOI: 10.1007/978-3-642-25405-5_13.

[10] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004. ISBN: 978-3-540-20854-9. URL: http://www.labri.fr/publications/l3a/2004/BC04.

[11] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. "Canonical Big Operators". In: *TPHOLs*. Ed. by Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar. Vol. 5170. Lecture Notes in Computer Science. Springer, 2008, pp. 86–101. ISBN: 978-3-540-71065-3. DOI: 10.1007/978-3-540-71067-7_11.

[12] M. Berz and K. Makino. "Rigorous global search using Taylor models". In: *SNC '09: Proceedings of the 2009 conference on Symbolic numeric computation*. Kyoto, Japan: ACM, 2009, pp. 11–20. ISBN: 978-1-60558-664-9. DOI: 10.1145/1577190.1577198.

[13] M. Berz, K. Makino, and Y-K. Kim. "Long-term stability of the Tevatron by verified global optimization". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 558.1 (2006). Proceedings of the 8th International Computational Accelerator Physics Conference - ICAP 2004, pp. 1 –10. ISSN: 0168-9002. DOI: 10.1016/j.nima.2005.11.035.

[14] Sandrine Blazy and Xavier Leroy. "Mechanized semantics for the Clight subset of the C language". In: *J. Autom. Reasoning* 43.3 (2009), pp. 263–288. URL: http://gallium.inria.fr/~xleroy/publi/Clight.pdf.

[15] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. "Full Reduction at Full Throttle". In: *CPP*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011, pp. 362–377. ISBN: 978-3-642-25378-2. DOI: 10.1007/978-3-642-25379-9_26.

[16] S. Boldo and M. Daumas. "Representable correcting terms for possibly underflowing floating point operations". In: *Proceedings of the 16th Symposium on Computer Arithmetic*. Ed. by J.-C. Bajard and M. Schulte. Santiago de Compostela, Spain: IEEE Computer Society Press, Los Alamitos, CA, 2003, pp. 79–86.

[17] S. Boldo, M. Daumas, C. Moreau-Finot, and L. Théry. *Computer Validated Proofs of a Toolset for Adaptable Arithmetic*. Tech. rep. Available at http://arxiv.org/pdf/cs.MS/0107025. École Normale Supérieure de Lyon, 2001.

[18] S. Boldo and G. Melquiond. "Emulation of FMA and correctly rounded sums: proved algorithms using rounding to odd". In: *IEEE Transactions on Computers* 57.4 (Apr. 2008), pp. 462–471.

[19] Sylvie Boldo. "Pitfalls of a Full Floating-Point Proof: Example on the Formal Proof of the Veltkamp/Dekker Algorithms". In: *IJCAR*. Ed. by Ulrich Furbach and Natarajan Shankar. Vol. 4130. Lecture Notes in Computer Science. Springer, 2006, pp. 52–66. ISBN: 3-540-37187-7. DOI: 10.1007/11814771_6.

[20] Sylvie Boldo and Marc Daumas. "A Simple Test Qualifying the Accuracy of Horner's Rule for Polynomials". In: *Numerical Algorithms* 37.1-4 (2004), pp. 45–60. DOI: 10.1023/B:NUMA.0000049487.98618.61.

[21] Sylvie Boldo and Marc Daumas. "Representable Correcting Terms for Possibly Underflowing Floating Point Operations". In: *IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, 2003, pp. 79–86. ISBN: 0-7695-1894-X. DOI: 10.1109/ARITH.2003.1207663.

[22] Sylvie Boldo and Guillaume Melquiond. "Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq". In: *IEEE Symposium on Computer Arithmetic*. Ed. by Elisardo Antelo, David Hough, and Paolo Ienne. IEEE Computer Society, 2011, pp. 243–252. ISBN: 978-0-7695-4318-5. DOI: 10.1109/ARITH.2011.40.

[23] Dan Boneh. "Finding Smooth Integers in Short Intervals Using CRT Decoding". In: *J. Comput. Syst. Sci.* 64.4 (2002), pp. 768–784. DOI: 10.1006/jcss.2002.1827.

[24] Pierre Boutillier. "A relaxation of Coq's guard condition". English. In: *Actes des Journées Francophones des langages Applicatifs*. Carnac, France, Feb. 2012, 14 pages. URL: http://hal.archives-ouvertes.fr/hal-00651780/en/.

[25] K. Briggs. *The* `doubledouble` *library*. Available at http://www.boutell.com/fracster-src/doubledouble/doubledouble.html. 1998.

[26] N. Brisebarre and S. Chevillard. "Efficient polynomial L^∞-approximations". In: *18th IEEE SYMPOSIUM on Computer Arithmetic*. Ed. by P. Kornerup and J.-M. Muller. Los Alamitos, CA: IEEE Computer Society, 2007, pp. 169–176.

[27] N. Brisebarre, J.-M. Muller, and A. Tisserand. "Computing Machine-efficient Polynomial Approximations". In: *ACM Transactions on Mathematical Software* 32.2 (2006), pp. 236–256.

[28] Nicolas Brisebarre, Mioara Joldeş, Érik Martin-Dorel, Micaela Mayero, Jean-Michel Muller, Ioana Paşca, Laurence Rideau, and Laurent Théry. "Rigorous Polynomial Approximation Using Taylor Models in Coq". In: *NASA Formal Methods 2012*. Ed. by Alwyn Goodloe and Suzette Person. Vol. 7226. Lecture Notes in Computer Science. Springer, 2012, pp. 85–99. ISBN: 978-3-642-28890-6. URL: http://hal-ens-lyon.archives-ouvertes.fr/ensl-00653460_v2/en/.

[29] Nicolas Brisebarre, Mioara Maria Joldeş, Peter Kornerup, Érik Martin-Dorel, and Jean-Michel Muller. "Augmented precision square roots, 2-D norms, and discussion on correctly rounding $\sqrt{x^2+y^2}$". In: *IEEE ARITH 2011*. Tuebingen, Germany, 2011, pp. 23–30. URL: http://hal-ens-lyon.archives-ouvertes.fr/ensl-00545591/en/.

[30] Francisco Cháves. "Utilisation et certification de l'arithmétique d'intervalles dans un assistant de preuves". French. PhD thesis. Lyon, France: École Normale Supérieure de Lyon, Sept. 2007. URL: http://tel.archives-ouvertes.fr/tel-00177109/en/.

[31] S. Chevillard. "Évaluation efficace de fonctions numériques. Outils et exemples". PhD thesis. Lyon, France: École Normale Supérieure de Lyon, 2009. URL: http://tel.archives-ouvertes.fr/tel-00460776/fr/.

[32] S. Chevillard, M. Joldeş, and C. Lauter. "Sollya: An Environment for the Development of Numerical Codes". In: *Mathematical Software - ICMS 2010*. Ed. by K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama. Vol. 6327. Lecture Notes in Computer Science. Heidelberg, Germany: Springer, Sept. 2010, pp. 28–31.

[33] Sylvain Chevillard, John Harrison, Mioara Joldeş, and Christoph Lauter. "Efficient and accurate computation of upper bounds of approximation errors". In: *Theoretical Computer Science* 16.412 (2011), pp. 1523–1543.

[34] Jacek Chrząszcz. "Implementing Modules in the Coq System". In: *TPHOLs*. Ed. by David A. Basin and Burkhart Wolff. Vol. 2758. Lecture Notes in Computer Science. Springer, 2003, pp. 270–286. ISBN: 3-540-40664-6. DOI: 10.1007/10930755_18.

[35] Jacek Chrząszcz. "Modules in Coq Are and Will Be Correct". In: *TYPES*. Ed. by Stefano Berardi, Mario Coppo, and Ferruccio Damiani. Vol. 3085. Lecture Notes in Computer Science. Springer, 2003, pp. 130–146. ISBN: 3-540-22164-6. DOI: 10.1007/978-3-540-24849-1_9.

[36] Cyril Cohen. "Types quotients en Coq". In: *Actes des 21èmes journées francophones des langages applicatifs (JFLA 2010)*. Ed. by Hermann. Vieux-Port La Ciotat, France: INRIA, Jan. 2010. URL: http://jfla.inria.fr/2010/actes/PDF/cyrilcohen.pdf.

[37] Pieter Collins, Milad Niqui, and Nathalie Revol. "A Taylor Function Calculus for Hybrid System Analysis: Validation in Coq". In: *NSV-3: Third International Workshop on Numerical Software Verification*. 2010.

[38] J. T. Coonen. "Underflow and the Denormalized Numbers". In: *Computer* 14 (1981), pp. 75–87. ISSN: 0018-9162. DOI: 10.1109/C-M.1981.220382.

[39] Don Coppersmith. "Finding a Small Root of a Bivariate Integer Equation; Factoring with High Bits Known". In: *EUROCRYPT*. Ed. by Ueli M. Maurer. Vol. 1070. Lecture Notes in Computer Science. Springer, 1996, pp. 178–189. ISBN: 3-540-61186-X. DOI: 10.1007/3-540-68339-9_16.

[40] Don Coppersmith. "Finding a Small Root of a Univariate Modular Equation". In: *EUROCRYPT*. Ed. by Ueli M. Maurer. Vol. 1070. Lecture Notes in Computer Science. Springer, 1996, pp. 155–165. ISBN: 3-540-61186-X. DOI: 10.1007/3-540-68339-9_14.

[41] Don Coppersmith. "Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities". In: *J. Cryptology* 10.4 (1997), pp. 233–260. DOI: 10.1007/s001459900030.

[42] M. Cornea, J. Harrison, C. Anderson, P. T. P. Tang, E. Schneider, and E. Gvozdev. "A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format". In: *IEEE Transactions on Computers* 58.2 (2009), pp. 148–162.

[43] Pascal Cuoq, Benjamin Monate, Anne Pacalet, and Virgile Prevosto. "Functional dependencies of C functions via weakest pre-conditions". In: *STTT* 13.5 (2011), pp. 405–417. DOI: 10.1007/s10009-011-0192-z.

[44] M. Daumas, G. Melquiond, and C. Muñoz. "Guaranteed proofs using interval arithmetic". In: *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*. Ed. by P. Montuschi and E. Schwarz. Cape Cod, MA, 2005, pp. 188–195.

[45] Marc Daumas, Laurence Rideau, and Laurent Théry. "A Generic Library for Floating-Point Numbers and Its Application to Exact Computing". In: *TPHOLs*. Ed. by Richard J. Boulton and Paul B. Jackson. Vol. 2152. Lecture Notes in Computer Science. Springer, 2001, pp. 169–184. ISBN: 3-540-42525-X. DOI: 10.1007/3-540-44755-5_13.

[46] Florent De Dinechin, Christoph Lauter, Jean-Michel Muller, and Serge Torres. "On Ziv's rounding test". English. Pre-print. 2012. URL: http://hal-ens-lyon.archives-ouvertes.fr/ensl-00693317/en/.

[47] T. J. Dekker. "A floating-point technique for extending the available precision". In: *Numerische Mathematik* 18.3 (1971), pp. 224–242.

[48] David Delahaye. "A Tactic Language for the System Coq". In: *LPAR*. Ed. by Michel Parigot and Andrei Voronkov. Vol. 1955. Lecture Notes in Computer Science. Springer, 2000, pp. 85–95. DOI: 10.1007/3-540-44404-1_7.

[49] David Delahaye and Micaela Mayero. "Dealing with algebraic expressions over a field in Coq using Maple". In: *J. Symb. Comput.* 39.5 (2005), pp. 569–592. DOI: 10.1016/j.jsc.2004.12.004.

[50] David Delahaye and Micaela Mayero. "Field, une procédure de décision pour les nombres réels en Coq". In: *JFLA*. Ed. by Pierre Castéran. Collection Didactique. INRIA, 2001, pp. 33–48. ISBN: 2-7261-1154-8.

[51] David Delahaye and Micaela Mayero. "Quantifier Elimination over Algebraically Closed Fields in a Proof Assistant using a Computer Algebra System". In: *Electr. Notes Theor. Comput. Sci.* 151.1 (2006), pp. 57–73. DOI: 10.1016/j.entcs.2005.11.023.

[52] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. "Assisted verification of elementary functions using Gappa". In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. Dijon, France, 2006, pp. 1318–1322. URL: http://www.lri.fr/~melquion/doc/06-mcms-article.pdf.

[53] N. I. Feldman and Yu. V. Nesterenko. *Transcendental numbers*. Vol. 44. Encyclopedia of mathematical sciences. Berlin: Springer-Verlag, 1998. ISBN: 3-540-61467-2.

[54] S. A. Figueroa. "A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages". PhD thesis. Department of Computer Science, New York University, 2000.

[55] S. A. Figueroa. "When is Double Rounding Innocuous?" In: *ACM SIGNUM Newsletter* 30.3 (July 1995).

[56] Jean-Christophe Filliâtre and Claude Marché. "The Why/Krakatoa/Caduceus Platform for Deductive Program Verification". In: *CAV*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 173–177. ISBN: 978-3-540-73367-6. DOI: 10.1007/978-3-540-73368-3_21.

[57] P. Friedland. "Algorithm 312: Absolute Value and Square Root of a Complex Number". In: *Communications of the ACM* 10.10 (Oct. 1967), p. 665.

[58] Ulrich Furbach and Natarajan Shankar, eds. *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Vol. 4130. Lecture Notes in Computer Science. Springer, 2006. ISBN: 3-540-37187-7.

[59] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. "Packaging Mathematical Structures". In: *Theorem Proving in Higher Order Logics*. Ed. by Tobias Nipkow and Christian Urban. Vol. 5674. Lecture Notes in Computer Science. Munich, Allemagne: Springer, 2009. URL: http://hal.inria.fr/inria-00368403/en/.

[60] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. 2nd ed. Cambridge University Press, 2003, xiv+785 pages.

[61] Herman Geuvers. "Inconsistency of classical logic in type theory". Short Note. Nov. 2007. URL: http://www.cs.ru.nl/~herman/PUBS/newnote.ps.gz.

[62] D. Goldberg. "What every computer scientist should know about floating-point arithmetic". In: *ACM Computing Surveys* 23.1 (Mar. 1991). An edited reprint is available at http://www.physics.ohio-state.edu/~dws/grouplinks/floating_point_math.pdf from Sun's Numerical Computation Guide; it contains an addendum *Differences Among IEEE 754 Implementations*, also available at http://www.validlab.com/goldberg/addendum.html., pp. 5–47.

[63] Georges Gonthier. "Formal Proof—The Four-Color Theorem". In: *Notices of the American Mathematical Society* 55.11 (2008), pp. 1382–1393. URL: http://www.ams.org/notices/200811/tx081101382p.pdf.

[64] Georges Gonthier and Assia Mahboubi. *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455. INRIA, 2008. URL: http://hal.inria.fr/inria-00258384/en/.

[65] Georges Gonthier and Assia Mahboubi. "An introduction to small scale reflection in Coq". In: *Journal of Formalized Reasoning* 3.2 (2010), pp. 95–152. URL: http://hal.inria.fr/inria-00515548/en/.

[66] S. Graillat, P. Langlois, and N. Louvet. "Algorithms for accurate, validated and fast computations with polynomials". In: *Japan Journal of Industrial and Applied Mathematics* 26.2 (2009), pp. 215–231.

[67] Benjamin Grégoire and Xavier Leroy. "A compiled implementation of strong reduction". In: *ICFP*. Ed. by Mitchell Wand and Simon L. Peyton Jones. ACM, 2002, pp. 235–246. ISBN: 1-58113-487-8. DOI: 10.1145/581478.581501.

[68] Benjamin Grégoire and Laurent Théry. "A Purely Functional Library for Modular Arithmetic and Its Application to Certifying Large Prime Numbers". In: *IJCAR*. Ed. by Ulrich Furbach and Natarajan Shankar. Vol. 4130. Lecture Notes in Computer Science. Springer, 2006, pp. 423–437. ISBN: 3-540-37187-7. DOI: 10.1007/11814771_36.

[69] Benjamin Grégoire, Laurent Théry, and Benjamin Werner. "A Computational Approach to Pocklington Certificates in Type Theory". In: *Functional and Logic Programming*. Ed. by Masami Hagiya and Philip Wadler. Vol. 3945. Lecture Notes in Computer Science. Springer, 2006, pp. 97–113. DOI: 10.1007/11737414_8.

[70] A. Griewank. *Evaluating Derivatives - Principles and Techniques of Algorithmic Differentiation*. SIAM, 2000.

[71] J. Harrison. "A Machine-Checked Theory of Floating-Point Arithmetic". In: *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*. Ed. by Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry. Vol. 1690. Lecture Notes in Computer Science. Nice, France: Springer-Verlag, Berlin, Sept. 1999, pp. 113–130.

[72] J. Harrison. "Formal verification of floating-point trigonometric functions". In: *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design, FMCAD 2000*. Ed. by W. A. Hunt and S. D. Johnson. Lecture Notes in Computer Science 1954. Springer-Verlag, Berlin, 2000, pp. 217–233.

[73] John Harrison. "HOL Light: A Tutorial Introduction". In: *FMCAD*. Ed. by Mandayam K. Srivas and Albert John Camilleri. Vol. 1166. Lecture Notes in Computer Science. Springer, 1996, pp. 265–269. ISBN: 3-540-61937-2. DOI: 10.1007/BFb0031814.

[74] John Harrison and Laurent Théry. "A Skeptic's Approach to Combining HOL and Maple". In: *J. Autom. Reasoning* 21.3 (1998), pp. 279–294. DOI: 10.1023/A:1006023127567.

[75] John Harrison and Laurent Théry. "Extending the HOL Theorem Prover with a Computer Algebra System to Reason about the Reals". In: *HUG*. Ed. by Jeffrey J. Joyce and Carl-Johan H. Seger. Vol. 780. Lecture Notes in Computer Science. Springer, 1993, pp. 174–184. ISBN: 3-540-57826-9. DOI: 10.1007/3-540-57826-9_134.

[76] John Harrison and Laurent Théry. "Reasoning About the Reals: The Marriage of HOL and Maple". In: *LPAR*. Ed. by Andrei Voronkov. Vol. 698. Lecture Notes in Computer Science. Springer, 1993, pp. 351–353. ISBN: 3-540-56944-8. DOI: 10.1007/3-540-56944-8_68.

[77] Kurt Hensel. "Neue Grundlagen der Arithmetik". In: *Journal für die reine und angewandte Mathematik (Crelle's Journal)* 1904.127 (1904). 10.1515/crll.1904.127.51, pp. 51–84.

[78] Y. Hida, X. S. Li, and D. H. Bailey. "Algorithms for Quad-Double Precision Floating-Point Arithmetic". In: *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*. Ed. by N. Burgess and L. Ciminiera. Vail, CO, June 2001, pp. 155–162. DOI: 10.1109/ARITH.2001.930115.

[79] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. 2nd ed. SIAM, Philadelphia, PA, 2002. ISBN: 0-89871-521-0.

[80] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[81] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. *The Coq proof assistant: a tutorial: version 8.3*. 2011. URL: http://coq.inria.fr/distrib/V8.3pl4/files/Tutorial.pdf.

[82] T. E. Hull, T. F. Fairgrieve, and P. T. P. Tang. "Implementing Complex Elementary Functions Using Exception Handling". In: *ACM Transactions on Mathematical Software* 20.2 (June 1994), pp. 215–244.

[83] Antonius J. C. Hurkens. "A Simplification of Girard's Paradox". In: *TLCA*. Ed. by Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin. Vol. 902. Lecture Notes in Computer Science. Springer, 1995, pp. 266–278. ISBN: 3-540-59048-X. DOI: 10.1007/BFb0014058.

[84] IEEE. "IEEE Standard for Binary Floating-Point Arithmetic". In: *ANSI/IEEE Std 754–1985* (1985). DOI: 10.1109/IEEESTD.1985.82928.

[85] IEEE. "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754–2008* (Aug. 2008). DOI: 10.1109/IEEESTD.2008.4610935.

[86] International Organization for Standardization. *Programming Languages – C*. Geneva, Switzerland: ISO/IEC Standard 9899:1999, Dec. 1999.

[87] C.-P. Jeannerod, N. Louvet, J.-M. Muller, and A. Panhaleux. "Midpoints and Exact Points of Some Algebraic Functions in Floating-Point Arithmetic". In: *IEEE Transactions on Computers* 60.2 (Feb. 2011). DOI: 10.1109/TC.2010.144.

[88] Mioara Joldeş. "Rigourous Polynomial Approximations and Applications". PhD thesis. Lyon, France: École Normale Supérieure de Lyon, 2011. URL: http://tel.archives-ouvertes.fr/tel-00657843/en/.

[89] Jean-Pierre Jouannaud and Zhong Shao, eds. *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011. ISBN: 978-3-642-25378-2.

[90] W. Kahan. "A logarithm too clever by half". Available at http://http.cs.berkeley.edu/~wkahan/LOG10HAF.TXT. 2004.

[91] W. Kahan. "Pracniques: further remarks on reducing truncation errors". In: *Commun. ACM* 8.1 (1965), p. 40. ISSN: 0001-0782. DOI: 10.1145/363707.363723.

[92] W. Kahan. *The Table-Makers' Dilemma and other Quandaries*. In Mathematical Software II, Informal Proceedings of a Conference, Purdue University, May 29–31, 1974. URL: http://books.google.fr/books?id=ff8hAQAAIAAJ.

[93] W. Kahan. *Why do we Need a Floating-Point Standard?* Tech. rep. Available at http://www.cs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf. Computer Science, UC Berkeley, 1981.

[94] William Kahan. "A Test for Correctly Rounded SQRT". Lecture note. May 1996. URL: http://www.cs.berkeley.edu/~wkahan/SQRTest.ps.

[95] A. Karatsuba and Y. Ofman. "Multiplication of Many-Digital Numbers by Automatic Computers". In: *Doklady Akad. Nauk SSSR* 145 (1962). Translation in Physics-Doklady 7, 595–596, 1963, pp. 293–294.

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

[97] D. Knuth. *The Art of Computer Programming*. 3rd. Vol. 2. Addison-Wesley, Reading, MA, 1998.

[98] Robbert Krebbers and Bas Spitters. "Computer Certified Efficient Exact Reals in Coq". In: *Calculemus/MKM*. Bertinoro, Italy, 2011, pp. 90–106.

[99] T. Lang and J.-M. Muller. "Bound on Run of Zeros and Ones for Algebraic Functions". In: *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*. Ed. by N. Burgess and L. Ciminiera. June 2001, pp. 13–20.

[100] V. Lefèvre. *The Euclidean Division Implemented with a Floating-Point Division and a Floor*. Research report RR-5604. INRIA, June 2005. URL: http://www.vinc17.org/research/papers/rr_intdiv.

[101] Vincent Lefèvre. "New Results on the Distance between a Segment and Z^2. Application to the Exact Rounding". In: *IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, 2005, pp. 68–75. ISBN: 0-7695-2366-8. DOI: 10.1109/ARITH.2005.32.

[102] Vincent Lefèvre and Jean-Michel Muller. "Worst Cases for Correct Rounding of the Elementary Functions in Double Precision". In: *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*. Ed. by Neil Burgess and Luigi Ciminiera. Vail, CO, June 2001, pp. 111–118. DOI: 10.1109/ARITH.2001.930115.

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

[104] A. K. Lenstra, H. W. Lenstra Jr., and L. Lovász. "Factoring polynomials with rational coefficients". In: *Mathematische Annalen* 261 (1982), pp. 515–534.

[105] Xavier Leroy. "A Formally Verified Compiler Back-end". In: *J. Autom. Reasoning* 43.4 (2009), pp. 363–446. DOI: 10.1007/s10817-009-9155-4.

[106] Xavier Leroy and Sandrine Blazy. "Formal verification of a C-like memory model and its uses for verifying program transformations". In: *J. Autom. Reasoning* 41.1 (2008), pp. 1–31. URL: http://gallium.inria.fr/~xleroy/publi/memory-model-journal.pdf.

[107] Pierre Letouzey. "A New Extraction for Coq". In: *Proceedings of the 2nd International Workshop on Types for Proofs and Programs (TYPES 2002)*. Ed. by Herman Geuvers and Freek Wiedijk. Vol. 2646. LNCS. Berg en Dal, Netherlands: Springer, 2003.

[108] P. Di Lizia. "Robust Space Trajectory and Space System Design using Differential Algebra". PhD thesis. Milano, Italy: Politecnico di Milano, 2008.

[109] Patrick Loiseleur. "Formalisation en Coq de la norme IEEE-754 sur l'arithmétique à virgule flottante". French. Rapport de stage de DEA. École Normale Supérieure de Lyon, 1997. URL: http://web.archive.org/web/20010605165747/http://www.lri.fr/~loisel/rapport-stage-dea.ps.gz.

[110] K. Makino. "Rigorous Analysis of Nonlinear Motion in Particle Accelerators". PhD thesis. East Lansing, Michigan, USA: Michigan State University, 1998.

[111] K. Makino and M. Berz. "Taylor Models and Other Validated Functional Inclusion Methods". In: *International Journal of Pure and Applied Mathematics* 4.4 (2003). http://bt.pa.msu.edu/pub/papers/TMIJPAM03/TMIJPAM03.pdf, pp. 379–456.

[112] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice-Hall, Englewood Cliffs, NJ, 2000.

[113] P. W. Markstein. "Computation of Elementary Functions on the IBM RISC System/6000 Processor". In: *IBM Journal of Research and Development* 34.1 (Jan. 1990), pp. 111–119.

[114] Ueli M. Maurer, ed. *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*. Vol. 1070. Lecture Notes in Computer Science. Springer, 1996. ISBN: 3-540-61186-X.

[115] Micaela Mayero. "Formalisation et automatisation de preuves en analyses réelle et numérique". PhD thesis. Université Paris VI, Dec. 2001. URL: http://www-lipn.univ-paris13.fr/~mayero/publis/these-mayero.ps.gz.

[116] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. "Certifying Algorithms". To appear in Computer Science Review. June 2010. URL: http://www.mpi-inf.mpg.de/~mehlhorn/ftp/CertifyingAlgorithms.pdf.

[117] G. Melquiond. "Floating-point arithmetic in the Coq system". In: *Proc. of the 8th Conference on Real Numbers and Computers*. 2008, pp. 93–102.

[118] Guillaume Melquiond. "Proving Bounds on Real-Valued Functions with Computations". In: *Proceedings of the 4th International Joint Conference on Automated Reasoning*. Sydney, Australia, Aug. 2008, pp. 2–17.

[119] Marc Mezzarobba. "Autour de l'évaluation numérique des fonctions D-finies". Thèse de doctorat. École polytechnique, Nov. 2011.

[120] P. Midy and Y. Yakovlev. "Computing some elementary functions of a complex variable". In: *Mathematics and Computers in Simulation* 33 (1991), pp. 33–49.

[121] Robin Milner. *Logic for Computable Functions: description of a machine implementation*. Tech. rep. Stanford University, May 1972. URL: ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/72/288/CS-TR-72-288.pdf.

[122] Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, eds. *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. Vol. 5170. Lecture Notes in Computer Science. Springer, 2008. ISBN: 978-3-540-71065-3.

[123] O. Møller. "Quasi Double-Precision in Floating-Point Addition". In: *BIT* 5 (1965), pp. 37–50.

[124] D. Monniaux. "The Pitfalls of Verifying Floating-Point Computations". In: *ACM TOPLAS* 30.3 (2008). A preliminary version is available at http://hal.archives-ouvertes.fr/hal-00128124, pp. 1–41.

[125] R. E. Moore. *Methods and Applications of Interval Analysis*. Society for Industrial and Applied Mathematics, 1979, xi+190 pages.

[126] J.-M. Muller. *Projet ANR TaMaDi – Dilemme du Fabricant de Tables – Table Maker's Dilemma (ref. ANR 2010 BLAN 0203 01)*. URL: http://tamadiwiki.ens-lyon.fr/tamadiwiki/.

[127] Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. 2nd ed. Birkhäuser Boston, MA, 2006. ISBN: 0-8176-4372-9.

[128] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010, 572 pages. ISBN: 978-0-8176-4704-9.

[129] César Muñoz, Víctor Carreño, Gilles Dowek, and Ricky Butler. "Formal verification of conflict detection algorithms". In: *International Journal on Software Tools for Technology Transfer* 4.3 (2003), pp. 371–380. DOI: 10.1007/s10009-002-0084-3.

[130] César Muñoz and David Lester. "Real number calculations and theorem proving". In: *18th International Conference on Theorem Proving in Higher Order Logics*. Oxford, England, 2005, pp. 239–254. DOI: 10.1007/11541868_13.

[131] Adam Naumowicz and Artur Kornilowicz. "A Brief Overview of Mizar". In: *TPHOLs*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 67–72. ISBN: 978-3-642-03358-2. DOI: 10.1007/978-3-642-03359-9_5.

[132] M. Neher, K. R. Jackson, and N. S. Nedialkov. "On Taylor Model Based Integration of ODEs". In: *SIAM J. Numer. Anal.* 45 (2007), pp. 236–262.

[133] A. Neumaier. "Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen". In: *ZAMM* 54 (1974). In German, pp. 39–51.

[134] A. Neumaier. "Taylor Forms – Use and Limits". In: *Reliable Computing* 9.1 (2003), pp. 43–79.

[135] Milad Niqui. "Formalising Exact Arithmetic: Representations, Algorithms and Proofs". PhD thesis. Radboud Universiteit Nijmegen, 2004.

[136] Ulf Norell. "Dependently Typed Programming in Agda". In: *Advanced Functional Programming*. Ed. by Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra. Vol. 5832. Lecture Notes in Computer Science. Springer, 2008, pp. 230–266. ISBN: 978-3-642-04651-3. DOI: 10.1007/978-3-642-04652-0_5.

[137] Russell O'Connor. "Certified exact transcendental real number computation in Coq". In: *Theorem Proving in Higher Order Logics*. 2008, pp. 246–261.

[138] T. Ogita, S. M. Rump, and S. Oishi. "Accurate Sum and Dot Product". In: *SIAM Journal on Scientific Computing* 26.6 (2005), pp. 1955–1988. ISSN: 1064-8275. DOI: 10.1137/030601818.

[139] Sam Owre, John M. Rushby, and Natarajan Shankar. "PVS: a prototype verification system". In: *11th International Conference on Automated Deduction*. Ed. by Deepak Kapur. Saratoga, New-York: Springer-Verlag, 1992, pp. 748–752. URL: http://pvs.csl.sri.com/papers/cade92-pvs/cade92-pvs.ps.

[140] Michael Parks. "Number-Theoretic Test Generation for Directed Rounding". In: *14th IEEE Symposium on Computer Arithmetic*. Adelaide, Australia, 1999, pp. 241–248.

[141] M. Pichat. "Correction d'une somme en arithmétique à virgule flottante". In: *Numerische Mathematik* 19 (1972). In French, pp. 400–406.

[142] Henri Poincaré. *La science et l'hypothèse*. Paris: Flammarion, 1902.

[143] Robert Pollack. "How to Believe a Machine-Checked Proof". In: *Twenty Five Years of Constructive Type Theory*. Ed. by G. Sambin and J. Smith. Oxford Univ. Press, 1998. URL: http://homepages.inf.ed.ac.uk/rpollack/export/believing.ps.gz.

[144] D. M. Priest. "Algorithms for arbitrary precision floating point arithmetic". In: *Proceedings of the 10th IEEE Symposium on Computer Arithmetic (Arith-10)*. Ed. by P. Kornerup and D. W. Matula. Grenoble, France: IEEE Computer Society Press, Los Alamitos, CA, June 1991, pp. 132–144.

[145] D. M. Priest. "On Properties of Floating-Point Arithmetics: Numerical Stability and the Cost of Accurate Computations". PhD thesis. University of California at Berkeley, 1992.

[146] E. Remez. "Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation (in French)". In: *C.R. Académie des Sciences, Paris* 198 (1934), pp. 2063–2065.

[147] S. M. Rump, T. Ogita, and S. Oishi. "Accurate Floating-Point Summation Part I: Faithful Rounding". In: *SIAM Journal on Scientific Computing* 31.1 (2008), pp. 189–224. DOI: 10.1137/050645671. URL: http://link.aip.org/link/?SCE/31/189/1.

[148] S. M. Rump, T. Ogita, and S. Oishi. "Accurate Floating-point Summation Part II: Sign, K-fold Faithful and Rounding to Nearest". In: *SIAM Journal on Scientific Computing* (2005–2008). Submitted for publication.

[149] D. M. Russinoff. "A Mechanically Checked Proof of Correctness of the AMD K5 Floating Point Square Root Microcode". In: *Formal Methods in System Design* 14.1 (1999), pp. 75–125.

[150] Amokrane Saïbi. "Typing Algorithm in Type Theory with Inheritance". In: *POPL*. 1997, pp. 292–301. DOI: 10.1145/263699.263742.

[151] B. Salvy and P. Zimmermann. "Gfun: a Maple package for the manipulation of generating and holonomic functions in one variable". In: *ACM Transactions on Mathematical Software* 20.2 (1994), pp. 163–177. DOI: 10.1145/178365.178368.

[152] E. M. Schwarz, M. M. Schmookler, and S. D. Trong. "FPU Implementations with Denormalized Numbers". In: *IEEE Transactions on Computers* 54.7 (2005), pp. 825–836. ISSN: 0018-9340. DOI: 10.1109/TC.2005.118.

[153] J. R. Shewchuk. "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates". In: *Discrete & Computational Geometry* 18 (1997), pp. 305–363. DOI: 10.1007/PL00009321.

[154] A. van der Sluis. "Upperbounds for roots of polynomials". In: *Numerische Mathematik* 15 (3 1970), pp. 250–262. ISSN: 0029-599X. DOI: 10.1007/BF02168974.

[155] Matthieu Sozeau and Nicolas Oury. "First-Class Type Classes". In: *TPHOLs*. Ed. by Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar. Vol. 5170. Lecture Notes in Computer Science. Springer, 2008, pp. 278–293. ISBN: 978-3-540-71065-3. DOI: 10.1007/978-3-540-71067-7_23.

[156] R. P. Stanley. "Differentiably Finite Power Series". In: *European Journal of Combinatorics* 1.2 (1980), pp. 175–188.

[157] D. Stehlé. "On the Randomness of Bits Generated by Sufficiently Smooth Functions". In: *Proceedings of the 7th Algorithmic Number Theory Symposium, ANTS VII*. Ed. by F. Hess, S. Pauli, and M. E. Pohst. Vol. 4078. Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2006, pp. 257–274.

[158] Damien Stehlé. "Algorithmique de la réduction des réseaux et application à la recherche de pires cas pour l'arrondi des fonctions mathématiques". PhD thesis. Université Nancy 1 Henri Poincaré, Dec. 2005.

[159] Damien Stehlé. "On the Randomness of Bits Generated by Sufficiently Smooth Functions". In: *Algorithmic Number Theory, 7th International Symposium, ANTS-VII, Berlin, Germany, July 23-28, 2006, Proceedings*. Ed. by Florian Hess, Sebastian Pauli, and Michael E. Pohst. Vol. 4076. Lecture Notes in Computer Science. Springer-Verlag, 2006, pp. 257–274. DOI: 10.1007/11792086_19.

[160] Damien Stehlé, Vincent Lefèvre, and Paul Zimmermann. "Searching Worst Cases of a One-Variable Function Using Lattice Reduction". In: *IEEE Transactions on Computers* 54.3 (Mar. 2005), pp. 340–346.

[161] P. H. Sterbenz. *Floating-Point Computation*. Englewood Cliffs, NJ: Prentice-Hall, 1974.

[162] *The Coq Proof Assistant: Reference Manual: version 8.3*. 2011. URL: http://coq.inria.fr/distrib/V8.3pl4/files/Reference-Manual.pdf.

[163] Laurent Théry and Guillaume Hanrot. "Primality Proving with Elliptic Curves". In: *Theorem Proving in Higher Order Logics*. Ed. by Klaus Schneider and Jens Brandt. Vol. 4732. Lecture Notes in Computer Science. Springer, 2007, pp. 319–333. DOI: 10.1007/978-3-540-74591-4_24.

[164] Freek Wiedijk. *The Seventeen Provers of the World*. Vol. 3600. Lecture Notes in Computer Science. Springer-Verlag, 2006.

[165] Chee-Keng Yap and Jihun Yu. "Foundations of Exact Rounding". In: *WALCOM*. Ed. by Sandip Das and Ryuhei Uehara. Vol. 5431. Lecture Notes in Computer Science. Springer, 2009, pp. 15–31. ISBN: 978-3-642-00201-4. DOI: 10.1007/978-3-642-00202-1_2.

[166] R. Zumkeller. "Formal Global Optimization with Taylor Models". In: *Proc. of the 4th International Joint Conference on Automated Reasoning*. 2008, pp. 408–422.

Abstract

We say that the Floating-Point (FP) implementation of a real-valued function is performed with correct rounding if the output is always equal to the rounding of the exact value. Requiring correct rounding for the implementation of standard mathematical functions has a number of advantages, including the reproducibility of numerical computations. But for implementing a function with correct rounding in a reliable and efficient manner, it is necessary to solve the so-called Table Maker's Dilemma (TMD). Two sophisticated algorithms (L and SLZ) have been designed to solve this problem, relying on some long and complex calculations (several years×CPU) that are performed by some heavily-optimized implementations. Hence the motivation to provide strong guarantees on these costly pre-computations.

To this end, we use the Coq formal proof assistant. First, we develop a formal library of Rigorous Polynomial Approximation (RPA) in Coq, a feature of which being the capability of computing an approximation polynomial as well as an interval that bounds the approximation error, in the proof assistant itself. This formalization constitutes a key building block for verifying the first step of SLZ, as well as the implementation of a mathematical function in general (with or without correct rounding).

Then we have implemented, formally verified and made effective three interrelated certificates checkers in Coq, whose correctness proof derives from Hensel's lemma that we have formalized for both univariate and bivariate cases. In particular, our "ISValP verifier" constitutes a key component for formally verifying the results generated by the SLZ algorithmic chain.

Then, we have focused on the mathematical proof of "augmented-precision" FP algorithms for the square root and the Euclidean 2D norm, whose IEEE 754-2008 standard recommends correct rounding. We give some tight lower bounds on the minimum non-zero distance between $\sqrt{x^2+y^2}$ and a "midpoint", which allows us to solve the TMD for this bivariate function.

Finally, the so-called "double-rounding" phenomenon can typically occur when several FP precision are available in a given architecture. Although it is, in general, innocuous, it may change the behavior of some usual small FP algorithms. In particular, we have formally verified a set of results describing the behavior of the Fast2Sum algorithm in presence of double-roundings, which led us to develop a Coq formalization on the notion of midpoints.

Keywords: floating-point arithmetic, IEEE 754 standard, correct rounding, Table Maker's Dilemma, SLZ algorithm, Coppersmith's technique, Coq formal proofs, SSReflect, Hensel's lemma, certificates, integral roots, rigorous polynomial approximation, Taylor models, interval arithmetic, Taylor–Lagrange remainder, D-finite functions, square root, 2D norms, midpoints, TwoMultFMA, Fast2Sum, TwoSum, double-rounding, Flocq library

Résumé

On dit que l'implantation en Virgule Flottante (VF) d'une fonction à valeurs réelles est réalisée avec arrondi correct si le résultat calculé est toujours égal à l'arrondi de la valeur exacte. Exiger l'arrondi correct pour l'implantation des fonctions mathématiques usuelles a de nombreux avantages, dont celui d'assurer la reproductibilité des calculs numériques. Mais pour implanter une fonction avec arrondi correct de manière fiable et efficace, il est nécessaire de résoudre ce qu'on appelle le dilemme du fabricant de tables (TMD en anglais). Deux algorithmes sophistiqués (L et SLZ) ont été conçus pour résoudre ce problème, en ayant recours à des calculs longs et complexes (plusieurs années×CPU) effectués par des implantations largement optimisées. D'où la motivation d'apporter des garanties fortes sur le résultat de ces pré-calculs coûteux.

Dans ce but, nous utilisons l'assistant de preuves formelles Coq. Tout d'abord nous développons une bibliothèque d'approximation polynomiale rigoureuse (RPA) en Coq, dont une caractéristique est de pouvoir calculer un polynôme d'approximation ainsi qu'un intervalle bornant l'erreur d'approximation à l'intérieur même de l'assistant de preuves. Cette formalisation constitue un élément clé pour valider la première étape de SLZ, ainsi que l'implantation d'une fonction mathématique en général (avec ou sans arrondi correct).

Puis nous avons implanté en Coq, formellement prouvé et rendu effectif trois vérifieurs de certificats, dont la preuve de correction dérive du lemme de Hensel que nous avons formalisé dans les cas univarié et bivarié. En particulier, notre « vérifieur ISValP » constitue un composant clé pour la certification formelle des résultats générés par l'algorithme SLZ.

Ensuite, nous nous sommes intéressés à la preuve mathématique d'algorithmes VF en « précision augmentée » pour la racine carré et la norme euclidienne en 2D, dont le standard IEEE 754-2008 recommande l'arrondi correct. Nous donnons des bornes inférieures fines sur la plus petite distance non nulle entre $\sqrt{x^2+y^2}$ et un « midpoint », ce qui nous permet de résoudre le dilemme du fabricant de tables pour cette fonction bivariée.

Enfin, lorsque différentes précisions VF sont disponibles dans une architecture donnée, peut survenir le phénomène de « double-arrondi ». Bien qu'en général inoffensif, il peut potentiellement changer le comportement de petits algorithmes usuels en arithmétique à VF. En particulier nous avons formellement prouvé un ensemble de théorèmes décrivant le comportement de l'algorithme Fast2Sum en présence de double-arrondis, ce qui nous a conduit à développer une formalisation Coq portant sur les midpoints.

Mots-clés : arithmétique à virgule flottante, standard IEEE 754, arrondi correct, dilemme du fabricant de tables, algorithme SLZ, technique de Coppersmith, preuves formelles Coq, SSReflect, lemme de Hensel, certificats, racines entières, approximation polynomiale rigoureuse, modèles de Taylor, arithmétique par intervalles, reste de Taylor–Lagrange, fonctions D-finies, racine carrée, norme 2D, midpoints, TwoMultFMA, Fast2Sum, TwoSum, double-arrondi, bibliothèque Flocq