INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A proof of GMP square root using the Coq assistant

Yves Bertot — Nicolas Magaud — Paul Zimmermann

N° 4475

June 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4475--FR+ENG

A proof of GMP square root using the Coq assistant

Yves Bertot*, Nicolas Magaud† , Paul Zimmermann‡

Thème 2 — Génie logiciel
et calcul symbolique
Projets Lemme et Spaces

Rapport de recherche n° 4475 — June 2002 — 28 pages

Abstract: We present a formal proof (at the implementation level) of an efficient algorithm proposed in [Zim99] to compute square roots of arbitrarily large integers. This program, which is part of the GNU Multiple Precision Arithmetic Library (GMP), is completely proven within the Coq system. Proofs are developed using the Correctness tool to deal with imperative features of the program. The formalization is rather large (more than 13000 lines) and requires some advanced techniques for proof management and reuse.

Key-words: arbitrary large numbers, formal methods, GMP, Coq

* Yves.Bertot@sophia.inria.fr
† Nicolas.Magaud@sophia.inria.fr
‡ Paul.Zimmermann@loria.fr

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 —Télécopie : +33 4 92 38 77 65

Preuve de la racine carrée de GMP dans l'assistant Coq

Résumé : Nous présentons une preuve formelle (au niveau de l'implantation) d'un programmme de calcul efficace de la racine carrée pour les grands entiers [Zim99]. Ce programme, qui fait partie de la bibliothèque de calcul en précision arbitraire GMP, est entièrement prouvé correct dans le système Coq. Les preuves sont réalisées en utilisant l'outil Correctness pour gérer les traits impératifs du programme. Le développement est assez gros (plus de 13 000 lignes) and requiert l'utilisation de techniques avancées de maintenance et de réutilisation des preuves.

Mots-clés : arithmétique en précision arbitraire, méthodes formelles, GMP, Coq

1 Introduction

Computing efficiently with multiple-precision integers requires different algorithms for different precisions. Usually, naïve schoolboy algorithms are used for small precision, and faster algorithms for larger precision. We consider here a fast algorithm computing the square root of an integer, which is efficient for both small and large precisions. This algorithm, presented at the abstract level in [Zim99], was implemented in the GMP library from version 4, and gave a speedup of about two with respect to the previous implementation.

We had two motivations in trying to prove the GMP square root implementation. Firstly, only the abstract (mathematical) level was proven to be correct, thus an implementation mistake was still possible; secondly, memory management of multiple-precision numbers provides challenging problems that were thus far not considered —to our best knowledge— with proof assistant techniques.

The Coq proof assistant was chosen since the theory of inductive constructions is well suited to proving recursive programs, as the square root implementation we consider here.

The main goal of this work is to show we can build a formal proof of a *real* program working on multiple-precision integers (here the square root publicly available with GMP) within a theorem prover such as Coq.

In the next paragraphs, we briefly present the GMP library and the Coq theorem prover.

1.1 GMP

The GNU Multiple Precision Arithmetic Library (GMP) [Gra02] is a library dealing with arithmetic operations on multiple-precision integers, rational and floating-point numbers. All its operations can be performed on arbitrarily large numbers and are designed to be as fast as possible. It is recognized as one of the best libraries to date.

This library has several layers. In addition to the high-level arithmetic functions on floating point numbers, it has a low-level function class. Functions belonging to this class have a name starting with the prefix `mpn`, and deal with non-negative integers represented as arrays of machine words (called *limbs* within GMP). For efficiency sake, most of them are directly written in assembly language (for instance `mpn_add_n` on x86 platforms). However the two functions we study in this paper, namely `mpn_sqrtrem` and `mpn_dq_sqrtrem`, are written in C.

1.2 Coq

Coq [Coq02] is a type-theoretic proof assistant. Its underlying logic is the calculus of inductive constructions [PM93] which is a typed λ-calculus with dependent types and capabilities for inductive definitions. Thanks to the Curry-Howard correspondence, building a proof p of a statement A can be viewed as building a term $\tilde{p}$ of type $\tilde{A}$. The main advantage of such a system is that proof terms can be typechecked automatically. It means once proved, a theorem can be automatically verified via type-checking its purported proof term.

This framework is well-suited to develop proofs of reliable software. In the following sections, our goal is to show how we take advantage of Coq capabilities to carry out formal descriptions of arithmetic programs.

1.3 Outline

In section 2, we first recall the basic technique to compute square roots and informally describe the algorithm we shall consider in this paper. We then study the algorithm at three different levels of abstraction. In section 3, we present the highest level of abstraction we consider to study the algorithm. In section 4, we show how to build a function extracting square roots. In section 5, we prove the correctness of the actual imperative implementation of the program. In section 6, we describe some related studies about formal proofs of arithmetic programs. In the conclusion, we sum up our contributions and give some guidelines for future work.

2 Square root extraction algorithms

We assume here that N is a nonnegative integer. An integer square root algorithm takes as input a nonnegative integer N and returns S and R, such that $S = \lfloor\sqrt{N}\rfloor$ and $R = N - S^2$. We call S the integer square root of N and R the corresponding remainder.

2.1 The schoolboy method

We recall the method formerly taught at school. As an example, we extract the square root of 7421. The hand-computing technique uses a division-like scheme.

$$\begin{array}{r|l} 7421 & 86 \\ \hline 1021 & 166 \cdot 6 = 996 \\ 25 & \end{array}$$

We list the steps of the extraction of a square root:

1. We split the input into blocks of two digits, here $n = 100n' + n''$ ($n' = 74$ and $n'' = 21$).
2. We compute the square root of the two most significant digits n'. Here we have $n' = 74$; this gives $s' = 8$ with $r' = 10$ as the remainder, we place 8 in the top right part of the figure and 10 below 74.
3. We lower the next two most significant digits n'' next to the remainder. This yields $100r' + n''$, here 1021.
4. We search for a digit x such that the expression p defined by $p = (2s' \cdot 10 + x) \cdot x$ is the greatest number less than or equal to $100r' + n''$. This gives $x = 6$.

5. We subtract p from $100r' + n''$, this yields the final remainder $r'' = 25$. The square root is $10s' + x$, *i.e.* 86.

If we consider the formula $(a+b)^2 - a^2 = (2a+b)b$, the fourth step of this algorithm is a way to subtract from the input the value of $(a+b)^2$ when one has already subtracted a^2.

The schoolboy method is a particular case of the algorithm we study in this paper (see section 3 with $L = \beta^2$ and $\beta = 10$).

2.2 The actual algorithm we study

The algorithm whose implementation `mpn_dq_sqrtrem` is proved in this paper uses one of the most common programming paradigms: the divide-and-conquer approach. It can be seen as an extension of the schoolboy method to the case where we consider blocks of $2n$ digits, instead of 2 digits. It can also be seen as a discrete version of Newton's method. The usual Newton iteration for the square root of a is:

$$x_{k+1} = x_k + \frac{a - x_k^2}{2x_k} = \frac{1}{2}(x_k + \frac{a}{x_k}).$$

Instead of computing with floating-point numbers, which requires recomputing a full division a/x_k at each step, the discrete version inductively updates the value of the remainder $a - x_k^2$. This algorithm can be used with any fast algorithms for multiplication and division, with a complexity of $\frac{4}{3}K(n)$ in the Karatsuba range and $6M(n)$ in the Fast-Fourier-Transform range [Zur94].

The algorithm [Zim99] takes as input a number N, splits it into two parts (N' and N''). It performs a recursive call on the most significant part N'. This yields a square root S' and a remainder R'. It recombines the remainder R' with the most significant half of N'' and divides by $2S'$. This yields an approximation for the square root. It then computes the associated remainder. The sign of this remainder determines whether the number computed so far is exactly the square root or an overestimation of one unit. That the overestimation is at most of one unit is one of the important mathematical properties of this algorithm. If needed, the algorithm finally corrects the square root and the remainder accordingly. We sketch its behaviour on the input 2703.

2703	52→ 51
203	20/10 = 2
-1	
102	

1. We split the input into two halves $n' = 27$ and $n'' = 03$.
2. We compute the square root and remainder for $n' = 27$; it yields $s' = 5$ and $r' = 2$. We place 2 below 27.

3. We lower one digit from n'', *i.e.* 0, and divide the number obtained, 20, by twice s' ($2s' = 10$). This yields $q = 2$ and $r'' = 0$. Hence the estimation of the square root is $10s' + q = 52$.

4. We determine the remainder of the square root extraction. For this, we subtract q^2 from $10r'' + 3$ (3 is the least significant digit of n''. This gives $3 - 2^2 = -1$ and we know that $2703 = 52^2 + (-1)$. The remainder is a negative number. From this we deduce we overestimated the square root.

5. Finally, we subtract one from the estimated square root. We get 51 and the remainder is increased accordingly by $2 \cdot 52 - 1$ to 102.

Below, we build a formal proof of the function `mpn_dq_sqrtrem`. Firstly we prove it at the propositional level, just considering which computations should be performed. Secondly, we build a functional program which actually computes the square root. Finally, we make a proof of the implementation at a very precise level of details.

3 Logical description of the algorithm

The first step of this study consists in proving the algorithm at a very abstract level. We start by describing how the algorithm works. All properties stated in this section have been formally proven using the Coq system. In this section, we assume the algorithm simply works on plain integers $\mathbb{Z}$. At this level, we do not bother with memory management or overflow issues. Such an algorithm can be specified as follows. It takes an integer $N > 0$ as input and returns two integers S and R such that:

$$\mathsf{SqrtremProp}(N, S, R) : \begin{cases} N = S^2 + R, \\ 0 < S, \\ 0 \leq R \leq 2S. \end{cases}$$

The property $R \leq 2S$ ensures that $N < (S+1)^2$, thus $S = \lfloor \sqrt{N} \rfloor$.

We have two distinct parts to the algorithm: the "core" algorithm, which has a normalization condition, and a wrapper. The core algorithm only computes square roots for a given range of values. The second part of the algorithm is simply a wrapper which checks whether this normalization condition holds for the input N. If it holds, it directly computes the square root via the core algorithm; otherwise, it first transforms the input N into a normalized number N_n. It then calls the core algorithm; this yields a square root S_n and a remainder R_n. It finally "denormalizes" these numbers into the actual square root S and remainder R for N.

3.1 The core algorithm

Let β be the basis, *i.e.* one more than the largest integer that can be encoded in a word.

The core algorithm takes as input a number $N \in \mathbb{N}$. This number is supposed to be normalized, *i.e.*:

$$\frac{\beta^{2n}}{4} \leq N < \beta^{2n}.$$

The first step of the algorithm is to decompose the input number into a lower part (least significant limbs) and a higher part (most significant limbs), corresponding to a division by $\beta^{2\lfloor \frac{n}{2} \rfloor}$. The actual divisor can be thought of in a more abstract manner and we just consider two integers H and L, where L plays the rôle of $\beta^{\lfloor \frac{n}{2} \rfloor}$. The integers H and L simply need to satisfy the following conditions:

$$H \geq L \tag{1}$$

$$L > 0 \tag{2}$$

and

$$HL = \beta^n. \tag{3}$$

We shall see later that H should also be even. The integers H and L will play a significant rôle throughout this description of the algorithm. The normalization condition for N can be expressed in terms of H and L as follows:

$$\frac{(HL)^2}{4} \leq N < (HL)^2.$$

In fact we prefer the following equivalent formulation, more adapted to our formal treatment of the proof:

$$N < (HL)^2 \leq 4N. \tag{4}$$

Computation steps. We now describe, step by step, the computations performed by the algorithm. We first break N down in two parts N' and N'', where N' simply is the most significant part and N'' the least significant part:

$$N = N'L^2 + N'', \qquad 0 \leq N'' < L^2.$$

The square root of N' is computed using a recursive application of the same algorithm and this returns values S' and R' for the square root and remainder of N' such that $\mathsf{SqrtremProp}(N', S', R')$ holds, that is:

$$(N' = S'^2 + R') \quad \wedge \quad (S' > 0) \quad \wedge \quad (0 \leq R' \leq 2S').$$

In the next step, N'' is broken down in two smaller parts N_1 and N_0:

$$N'' = N_1 L + N_0, \qquad \text{with } 0 \leq N_0, N_1 < L,$$

and $R'L + N_1$ is divided by $2S'$. This yields a quotient Q and a remainder R'':

$$R'L + N_1 = Q(2S') + R'', \qquad 0 \leq R'' < (2S').$$

All these computations lead to the following result:

$$\begin{aligned} N &= N'L^2 + N_1 L + N_0 \\ &= (S'^2 + R')L^2 + N_1 L + N_0 \\ &= (S'L)^2 + (R'L + N_1)L + N_0 \\ &= (S'L)^2 + (2S'Q + R'')L + N_0 \\ &= (S'L + Q)^2 + (R''L + N_0 - Q^2). \end{aligned} \tag{5}$$

These equations suggest that $(S'L + Q)$ might be the integer square root of N. We shall see that this number is very close to the actual square root. Actually it is either the square root or an overestimation by at most one. Central properties to establish this result are the facts $L \leq 2S'$ and $Q \leq L$.

A lower bound on the recursive square root S'. The normalization property, the equation defining N' and N'', and the bound on N'' ensure the following inequalities:

$$H^2L^2 \leq 4(N'L^2 + N'') < 4(N' + 1)L^2.$$

Assume H is even. We can consider $H' = \frac{H}{2}$ and simplify the above inequality into

$$H'^2 < N' + 1$$

which is equivalent to

$$H^2 \leq 4N'.$$

Moreover, it is trivial to deduce that N' is necessarily smaller than H^2, so that the preconditions for a recursive call of the algorithm are satisfied. Since S' is the square root of N' and $L \leq H$, this yields a useful inequality for S':

$$L \leq 2S'. \tag{6}$$

An upper bound on the quotient Q. The number Q is constructed by a division so that $2S'Q + R'' = R'L + N_1$ and R' is obtained through the recursive square root computation, so that $R' \leq 2S'$. Since $N_1 < L \leq 2S'$ we obtain

$$2S'Q + R'' < 2S'(L + 1).$$

Knowing that R'' is positive, this gives

$$Q \leq L. \tag{7}$$

No underestimation of the square root. We show here that $S'L+Q$ is not an underestimation of the square root. It suffices to prove that $S'L+Q+1$ is an overestimation, in other words $N < (S'L+Q+1)^2$. This is equivalent to the property that $N-(S'L+Q)^2 \leq 2(S'L+Q)$ and using equation (5), to the inequality :

$$R''L + N_0 - Q^2 \leq 2(S'L + Q).$$

We know that $R'' \leq 2S' - 1$ and $N_0 < L$, because R'' and N_0 are remainders of divisions by $2S'$ and L. From this we can derive the following inequality:

$$R''L + N_0 \leq (2S' - 1)L + L = 2S'L$$

All this combined with the fact that Q is positive fits in the following sequence of comparisons:

$$R''L + N_0 - Q^2 \leq R''L + N_0 \leq 2S'L \leq 2(S'L + Q).$$

A limit on overestimation. The candidate $S'L+Q$ may still be an overestimation. If so, then $S'L + Q - 1$ is another candidate for the square root and the corresponding remainder is

$$R''L + N_0 - Q^2 + 2(S'L + Q) - 1. \tag{8}$$

We will simply show that this expression is always nonnegative. First the lower bound (6) on $2S'$ and the upper bound (7) on Q yield the sequence of inequalities

$$Q - 1 \leq L \leq 2S'.$$

So we know separately $Q - 1 \leq 2S'$ and $Q - 1 \leq L$. If $Q \geq 1$, we deduce the following inequality:

$$(Q - 1)^2 \leq 2S'L,$$

which also holds for $Q = 0$ since S' and L are positive integers. This inequality can be rewritten as follows:

$$0 \leq 2(S'L + Q) - Q^2 - 1.$$

As N_0, L, and R'' are non-negative, we can add $R''L + N_0$ to the right hand side, which shows that the corrected remainder (8) is non-negative.

Altogether, the last two paragraphs express that the square root is either $S'L + Q$ or $S'L+Q-1$. To tell which is the right value, we only need to compute the sign of the remainder $R''L + N_0 - Q^2$. If this remainder is non-negative, we already know that $R''L + N_0 - Q^2 \leq 2(S'L+Q)$, and both constraints on the remainder are fulfilled. If the remainder is negative, then we add $2(S'L + Q) - 1$ to it, which necessarily yields a non-negative value. It remains to check that this new value is smaller or equal to $2(S'L + Q - 1)$; this is immediate since this value is obtained by adding a strictly negative value to $2(S'L + Q) - 1$.

Relating H, L, and the basis β. With respect to the basis β, the normalization condition imposes

$$\frac{\beta^{2n}}{4} \leq N < \beta^{2n}.$$

The numbers H and L simply made it possible to abstract away from the basis during the abstract reasoning phase. However, we have to understand how these numbers are produced and how the constraints put on them can be interpreted as constraints on the basis.

First, we need $HL = \beta^n$ and $L \leq H$. Taking $L = \beta^l$ and $H = \beta^{n-l}$ with $l \leq \frac{n}{2}$ works, for example

$$l = \lfloor \frac{n}{2} \rfloor, \qquad L = \beta^l, \qquad H = \beta^{n-l}.$$

For recursive calls, this ensures that the length of the number decreases at each call, except when $n = 1$. Concretely, this means that this algorithm requires another algorithm to compute the square root of integers N with $\frac{\beta^2}{4} \leq N < \beta^2$. Such a function is actually provided in the GMP library. We haven't certified this function and have simply assumed that it satisfies the specification SqrtremProp.

The last constraint that we have seen in our abstract proof is that H should be even. If $n = 2$ this leads to the constraint that β should be even. On the other hand, if β is even, we are sure that $H = \beta^h$ is too. In practice, this means that this algorithm could be used to compute square roots for numbers written in arbitrary even bases, for instance basis 10.

The assumption H even helped to show $L \leq 2S'$, which in turn implied $Q \leq L$, and both inequalities were used to show that the corrected remainder (8) is always nonnegative, therefore at most one correction is needed.

Our algorithm simply does not work if H is odd. Assume H odd. The above bounds (6) and (7) only fail when $L = H = 2S' + 1$ and $Q = L + 1$, with input $\frac{1}{4}L^4 + \frac{1}{2}L^3 + \frac{1}{4}L^2 - L$. Consider for example $H = L = 3$, and $N = 33$. The first square root and remainders are $(7, -16)$, then the corrected ones are $(6, -3)$, thus two corrections are required in that case.

Formal proof development. The description of the algorithm justification is as close as possible without going into unruly details to the formal proof we have done in the computer. A major difference is the order of reasoning steps. When performing proofs on the computer, it is customary to reason in backward manner: start from the goal proposition, propose a new formulation that is strong enough to imply the initial goal, but simpler, and break down into pieces until all can be verified automatically by the computer.

Aside from this difference, we have followed the same abstraction steps: the basis β does not appear in the main proof and everything is expressed in terms of the various intermediate results of computations, N, N', S', R', N_1, Q, etc. The main proof fits in a 600 line long script that can be replayed step by step on the computer to see the evolution of the results. Proofs can usually be broken down in several files and our proof uses two other auxiliary files adding general facts about integer arithmetic (about 150 lines) and the power function (about 150 lines). We have used the Coq [ABPR01] proof development tool to perform this proof, thus benefiting from its capabilities to render mathematical formulas with traditional mathematical notations, especially for the square and power functions, that use superscripts.

3.2 The wrapper

The user interface function to compute square roots does not have a precondition imposing that the input should be normalized or its length should be even. In practice all numbers are given with their length, that is we receive both a number N as input and a number n such that $0 < N < \beta^n$.

If N satisfies the normalization condition, *i.e.*, if n is even and $\frac{\beta^n}{4} \leq N < \beta^n$, its square root can be computed via the core algorithm. Otherwise, a wrapping procedure transforms N into a suitable input for the core algorithm, performs the computation with the core algorithm, and then interprets the result to yield the required value.

From now on, we assume that the basis is an even power of 2 and we define b to be the number of bits per limb, that is $\beta = 2^b$. Let $2c$ or $2c+1$ be the number of leading zero bits in the input's most significant limb and let $t = \lceil \frac{n}{2} \rceil$. If n is even and $c = 0$, then the input is normalized and can be passed directly to the core algorithm. The results do not need further treatment.

If n is odd or $c > 0$, then we build a new number $N_1 = 2^{2c}\beta^{2t-n}N$. This new number is normalized with respect to β^{2t}. Please note that $2t - n$ is either 0 or 1.

Let $k = c + (2t - n)b/2$. We have $N_1 = 2^{2k}N$. As N_1 is a normalized number with respect to β^{2t}, we can compute two numbers S_1 and R_1 such that

$$N_1 = S_1^2 + R_1, \qquad S_1 > 0, \qquad 0 \leq R_1 \leq 2S_1. \tag{9}$$

Writing $S_1 = 2^k S + s_0$ with $0 \leq s_0 < 2^k$, we have:

$$\begin{aligned} {S_1}^2 &\leq 2^{2k}N < (S_1 + 1)^2 \\ (2^k S + s_0)^2 &\leq 2^{2k}N < (2^k S + s_0 + 1)^2 \\ 2^{2k}S^2 &\leq 2^{2k}N < (2^k S + 2^k)^2 \\ 2^{2k}S^2 &\leq 2^{2k}N < 2^{2k}(S+1)^2. \end{aligned}$$

Dividing by 2^{2k} leads to $S^2 \leq N < (S+1)^2$, thus S is the integer square root of N. From it, we easily prove $0 \leq R \leq 2S$.

4 Functional description of the core algorithm

The abstract description of the algorithms given in the previous sections only makes it possible to express that the relations linking all intermediate values ensure that the final results satisfy the specification. However, they do not describe a program and, in particular, they leave the possibility for errors with respect to the required properties of the input for each function or recursive call. For a complete description of the algorithm, we need to show that this algorithm can actually be represented by a function in the programming language that is embedded in our formal framework's language.

This functional description is going to rely heavily on *dependent types* in two manners, to ensure that each function is called with input satisfying the conditions required for their good behavior and termination.

First, dependent types are used to express the restrictions on the input for functions. This is done by giving extra arguments to functions, that are used as certificates that the data satisfies the required properties. For instance, we use a function `decompose` that receives a value of type $\mathbb{N}$, which should be greater than 1. To express this, we say that the function `decompose` has a type with the following form:

$$\Pi n : \mathbb{N}.(1 < n) \to \ldots$$

This type indicates that the function takes two arguments, where the first one is a natural number n and the second argument is a certificate ensuring that n is greater than 1. The proposition $1 < n$ actually is the type of the certificate, and this means that the type of the second argument depends on the first argument. This is why we say that we use dependent types. The proof assistant that we have used is based on this notion of dependent types: it belongs to the family of type theory based proof systems. The trick of type theory is to consider that types can be read alternatively as data types or as propositions. Correspondingly, elements of the types can be viewed as data structures or as proofs. In this sense, most theorems make it possible to transform certificates for some requirements into certificates for other requirements. In practice, the Π and $\to$ notations given above correspond exactly to the quantifier $\forall$ and the implication, respectively, when one considers logical statements. For this reason, we shall always use the $\forall$ symbol to express that a function takes an argument as input, such that the output or later arguments have a type depending on this argument.

Functions computing data can also produce certificates attached to the data. For instance, the `decompose` function produces a pair of natural numbers, h and l, such that:

$$h + l = n \quad \wedge \quad h < n \quad \wedge \quad 0 < l \leq h.$$

Obviously, these properties can only be satisfied if $1 < n$. This explains why the `decompose` function needs such an input certificate. Altogether, the type of the `decompose` function could be read as the following one:[1]

$$\texttt{decompose} : \Pi n : \mathbb{N}.(1 < n) \to \{h : \mathbb{N} \& \{l : \mathbb{N} \mid l + h = n \wedge h < n \wedge 0 < l \wedge l \leq h\}\}.$$

For the abstract description of the algorithm we have used two numbers H and L. This description imposes only simple constraints on these two numbers, and their relation to the actual basis for computation does not appear in the formal proof. Actually, we are going to express that H and L are powers of the basis and we are going to use the `decompose` function given above to produce these two numbers. The core algorithm is thus going to accept only numbers that are normalized with respect to an even power of the basis. To make notation more convenient, we define the property IsNormalized as follows:

$$(\mathsf{IsNormalized}\ n\ v) \equiv n < v \leq 4n.$$

[1] We actually use a different formulation that would be too long to describe here.

To express that the number n needs to be normalized with respect to some even power of the basis, we simply say that there exists an h such that the property (IsNormalized n β^{2h}) holds. In simpler terms, h is half of the number of digits of the input and the predicted number of digits for the output.

The number of digits is also needed in the algorithm, so that the function representing the core algorithm has the following type:

$$\forall h : \mathbb{N}.\ \forall n : \mathbb{Z}.\ (\mathsf{IsNormalized}\ n\ \beta^{2h}) \rightarrow \{s : \mathbb{Z}\ \&\ \{r : \mathbb{Z} \mid (\mathsf{SqrtremProp}\ n\ s\ r)\}\}.$$

This is the type of a function taking three arguments: a natural number h an integer n as regular data, and a certificate ensuring that h and n are consistent in some sense. The result is a compound result again containing three pieces of data: two integers s and r and a proof that these integers satisfy the specification that we have studied in the previous section.

To simplify the notation in the rest of this paper, we define an abbreviation for a fragment of this type:

$$\begin{aligned}(\mathsf{sqrt\_F\_type}\ h) \quad \equiv \quad & \forall n : \mathbb{Z}.\quad (\mathsf{IsNormalized}\ n\ \beta^{2h}) \rightarrow \\ & \{s : \mathbb{Z}\ \&\ \{r : \mathbb{Z} \mid (\mathsf{SqrtremProp}\ n\ s\ r)\}\}.\end{aligned}$$

Regular data arguments and proof arguments are passed from one function to another with the help of pattern matching constructs. For instance, the call to the `decompose` function in our functional description appears in a fragment with the following shape

```
Cases (Zle_lt_dec h 1) of
  (left H_h_le_1) ⇒ ...
| (right H_1_lt_h) ⇒
  Cases (decompose h ?) of
    (h', l, Heqh, H_h'_lt_h, H_O_lt_l, H_l_le_h) ⇒ ...
```

This represents a call to the function `decompose` on the value h; its second argument, which should be a certificate that h is greater than 1, is left undescribed: the proof system constructs a proof obligation that will be proven later by the programmer. In this case, this proof is trivial to provide, since it has to be constructed under the assumption $H\_1\_lt\_h$ that was provided by the test function `Zle_lt_dec`: this function not only compares h and 1, it also returns certificates and $H\_1\_lt\_h$ is a certificate that can be used as a proof that $1 < h$. Thus, the proof can be directly $H\_1\_lt\_h$ here. In other cases, more complex proofs may need to be constructed. In turn, the results of `decompose` can be viewed as two natural numbers h' and l, a proof that $h' + l = h$, a proof that $h' < h$, a proof that $0 < l$ and a proof that $l \leq h'$.

To describe recursive functions, proof arguments also need to be provided to ensure that the function terminates. In practice, a recursive function of type $A \rightarrow B$ can be described by a function of type

$$\forall x : A.(\forall y : A.(R\ y\ x) \rightarrow B) \rightarrow B,$$

where R is a *well-founded* relation, that is, a relation that has no infinite decreasing chain. This function has two arguments, and the second argument corresponds to recursive calls. Its type expresses that recursive calls can only be made on values that are smaller than the initial argument (where R is the relation "...is smaller than ..."). The absence of infinite chains then ensures that the recursive function is eventually going to terminate. For our core algorithm, we use $\mathbb{N}$ as the input type instead of A and the natural strict order $<$ on $\mathbb{N}$ as the well-founded relation. The recursive algorithm is then represented by the function sqrt_F with the following type

$$\forall h : nat.\quad (\forall h' : \mathsf{nat}.\ h' < h \to (\mathsf{sqrt\_F\_type}\ h')) \to (\mathsf{sqrt\_F\_type}\ h).$$

So recursive calls of the algorithm can only happen on values h' that are smaller than the input value h, but we have already seen that the `decompose` function given above will make it possible to produce such values, with the relevant certificate, when h is large enough.

The main structure of the sqrt_F function is given by the following expression:

$\lambda h, sqrt, n, Hnorm.$
Cases (`le_lt_dec` h 1) of
  (left Hhle) ⇒ (normalized_base_case h ? n ?)
| (right H1lth) ⇒
  Cases (`decompose` h ?) of
    (h', l,$Heqh$, $H\_h'\_lt\_h$, $H\_O\_lt\_l$, $H\_l\_le\_h$) ⇒
    Cases (`div` n β^{2l} ? ?) of
      (n', n'', $Hdiv$) ⇒
      Cases (`div` n'' β^{l} ? ?) of
        (n_1, n_0, $Hdiv'$) ⇒
        Cases ($sqrt\ h'$? n' ?)of
          (s', (r', $Hsqrtrem$)) ⇒
          Cases (`div` $r'\beta^l + n_1$ $2s'$? ?) of
            (q, r'', $Hdiv_1$) ⇒
            Cases (`Z_le_gt_dec` 0 $(r''\beta^l + n_0 - q^2)$) of
              (left H_0_le_R) ⇒ ($s'\beta^l + q$,($r''\beta^l + n_0 - q^2$, ?))
            | (right HltR) ⇒
              ($s'\beta^l + q - 1$,($r''\beta^l + n_0 - q^2 + 2(s'\beta^l + q) - 1$, ?)
...

All the question marks in this text correspond to proof obligations that can be verified with the proof assistant in the same manner as for regular theorems. Notice that the recursive call to the algorithm is represented by the expression ($sqrt\, h'$? n' ?). This recursive call itself requires two proof obligations, one to ensure that h' is strictly smaller than h and the other to ensure that n' is normalized with respect to $\beta^{2h'}$.

5 The actual imperative program

In the previous two sections, we do not consider memory management issues. Actually, the traditional point of view in functional programming is that memory should be managed automatically with the help of a garbage collector, so that all notions of memory are abstracted away and the user has no control on how efficiently the memory is actually managed. This is unfortunate in our case, because the actual implementation is designed in a very careful way to optimize memory consumption. This care means opportunities for errors and a good opportunity for formal verification.

The main characteristic of this implementation is that it works *in-place*. The input number requires $2n$ limbs of memory storage, and the square root and remainder also require $2n + O(1)$ limbs altogether, however the algorithm makes sure that only $3n$ limbs are used to store all intermediate results. As a consequence, the input is actually destroyed during the computation and the remainder is actually stored in the lower part of the input. The exact place where the remainder is kept is important for the algorithm's correct operation.

5.1 The C procedure

We give in Fig. 1 the exact C code of the procedure as it can be found in GMP 4.0.1 [Gra02]. The careful reader will notice the similarity with the algorithm described in the previous section and also some important differences. First, the number that is called h in the previous sections is called n here. Then there is no need to compute the numbers $H = \beta^h$ or $L = \beta^l$ or the numbers N', N'', N_1, and N_0, since all these numbers already lie at some place in memory. If N is the number using $2n$ limbs and whose least significant limb is at location n_p, then N' is the number of length $2h$ whose least significant limb is at location $n_p + 2l$, N_1 has length l and sits at location $n_p + l$ and N_0 sits at location n_p, without requiring any extra computation. Similarly, if one computes the square root of N' so that the remainder R' is placed in memory at location $n_p + 2l$ (with length h) and N_1 is at location $n_p + l$ (with length l), then without any computation one obtains $R'L + N_1$ at location $n_p + l$, with length $h + l = n$. This trick is used at line 14 for $R'L + N_1$ and at line 21 for $R''L + N_0$.

Secondly, most binary operations on large numbers actually take 4 arguments: two arguments indicate where the lowest limb of the input numbers lie in memory (these arguments are pointers), one argument indicates where the lowest limb of the output will lie after the computation, and the last argument indicates the (common) length of the arguments. For instance `mpn_sub_n` is the function used to subtract a number of n limbs from another number of n limbs and the result is a number of n limbs. Operations like subtraction and addition may provoke an underflow or an overflow, usually corresponding to an extra bit that is the value returned by the function.

The square root function may itself incur an overflow. If one computes the square root of a number that fits in $2n$ limbs, the square root itself is sure to fit in n limbs; however, one only knows that the remainder is smaller or equal to twice the root, and for this reason

```
1  int mpn_dq_sqrtrem (mp_ptr sp, mp_ptr np, mp_size_t n) {
2    mp_limb_t q; /* carry out of {sp, n} */
3    int c, b; /* carry out of remainder */
4    mp_size_t l, h;
5
6    ASSERT (np[2*n-1] >= MP_LIMB_T_HIGHBIT/2);
7
8    if (n == 1) return mpn_sqrtrem2(sp, np, np);
9
10   l = n / 2;
11   h = n - l;
12   q = mpn_dq_sqrtrem (sp + l, np + 2 * l, h);
13   if (q) mpn_sub_n (np + 2 * l, np + 2 * l, sp + l, h);
14   q += mpn_divrem (sp, 0, np + l, n, sp + l, h);
15   c = sp[0] & 1;
16   mpn_rshift (sp, sp, l, 1);
17   sp[l-1] |= q << (BITS_PER_MP_LIMB - 1);
18   q >>= 1;
19   if (c) c = mpn_add_n (np + l, np + l, sp + l, h);
20   mpn_sqr_n (np + n, sp, l);
21   b = q + mpn_sub_n (np, np, np + n, 2 * l);
22   c -= (l == h) ? b : mpn_sub_1 (np + 2 * l, np + 2 * l, 1, b);
23   q = mpn_add_1 (sp + l, sp + l, h, q);
24
25   if (c < 0) {
26       c += mpn_addmul_1 (np, sp, n, 2) + 2 * q;
27       c -= mpn_sub_1 (np, np, n, 1);
28       q -= mpn_sub_1 (sp, sp, n, 1);
29     }
30   return c;
31 }
```

Figure 1: The `mpn_dq_sqrtrem` function as distributed with GMP 4.0.1 (only line breaks have been edited from the actual code).

it may not fit in n limbs and have a one-bit overflow. This one-bit overflow is the value returned by the function.

Third, there is no division of $R'L + N_1$ by $2S'$ as in §3.1. Such a division would require computing first the number $2S'$ and finding a place in memory to store this intermediate value. Rather, a first step is to divide by S' (line 14) and then to divide the quotient by 2 (line 16), updating the remainder if the quotient is odd (line 19). Dividing by S' is a bit tricky because of the way R' is stored in memory. The whole process appears between lines 13 and 19 (included), and we will describe it more carefully in section 5.4.1.

5.2 CORRECTNESS: developing formal proofs of imperative programs

CORRECTNESS [Fil99, Fil01] is a tool designed to carry out proofs of imperative programs within the COQ proof assistant. It takes as input a program written in a ML-like language with imperative features and its specification as a logical formula, annotated in a Floyd-Hoare style. For this algorithm, ML-like semantics are close enough to the behavior of the C programming language and we have used this tool in our study. The tool produces a collection of verification conditions, for which the user is required to provide proofs. These proofs can then be verified using COQ.

In our model of the C program, memory is represented as a global array m of size bound. Its indices range from 0 to $\mathsf{bound} - 1$. We replaced pointers by indices in this memory array and lengths are computed as regular natural numbers. Each cell in the array is supposed to represent a word in memory (a limb in GMP parlance). For our formalization, we have introduced a type $\mathsf{mod}\mathbb{Z}$ with the following definition:

$$\mathsf{mod}\mathbb{Z} \equiv \{v : \mathbb{Z} \mid 0 \leq v < \beta\}.$$

Thus, memory cells can only be used to store bounded nonnegative integers and we also provide a function $\mathsf{mod}\mathbb{Z}\_\mathsf{to}\_\mathbb{Z}$ to inject elements of $\mathsf{mod}\mathbb{Z}$ in $\mathbb{Z}$.

5.2.1 Interpreting memory segments

We start by defining an interpretation function: $m, pos, l \mapsto \{pos, l\}_m$. It is intended to return the actual integer encoded in a memory segment. It takes three parameters as input: the array denoting the memory m, the position pos where the segment starts, and its length l. Here, pos and l are natural numbers, so that we can easily perform case analysis and induction on them within the COQ system. The interpretation function is defined by the two equations:

$$\begin{aligned} \{pos, 0\}_m &= 0, \\ \{pos, l+1\}_m &= \mathsf{mod}\mathbb{Z}\_\mathsf{to}\_\mathbb{Z}(m[pos]) + \beta\{pos+1, l\}_m. \end{aligned}$$

This function is useful to describe formally the basic operations of the GMP library and to connect variables used in §3 to values stored in memory.

The property that placing two numbers a and b side-by-side in memory is enough to compute a new number of the form $a\beta^l + b$ is described in our formal development by the following theorem, which we prove by induction on l:

$$\mathsf{lmpnumber\_decompose} : \forall m, h, l, p.\ \{p, l\}_m + \beta^l \{p + l, h\}_m = \{p, l + h\}_m.$$

5.2.2 Describing basic GMP functions

CORRECTNESS provides means to declare signatures (or interfaces) for functions without defining them. For instance the subtraction function mpn_sub_n can be described as follows:

```
Global  Variable mpn_sub_n :
        fun (pos_r : nat)(pos_a : nat)(pos_b : nat)(l : nat)
        returns _ : unit
        reads m
        writes m, rb
        post {pos_r, l}_m = (rb · β^l + {pos_a, l}_m@) − {pos_b, l}_m@
end.
```

In other words, mpn_sub_n takes as input pos_r, pos_a, pos_b, and l, returns nothing and may have some side-effects on the memory m and the boolean reference rb. The input numbers are stored in $\{pos_a, l\}$ (first argument) and $\{pos_b, l\}$ (second argument), and the output is stored in $\{pos_r, l\}$. In this description there is no pre-condition[2]. A minimal post-condition states that it subtracts the number laying in the initial memory (denoted by $m@$) at position pos_b and of length l from the number also laying in the initial memory $m@$ at position pos_a and of length l. It writes the result in m at position pos_r with the exception of the borrow (0 or 1) that is stored in the global variable rb. That variable will actually be set when the second argument is larger than the first, so the result should be negative. The result is made positive again by adding $rb \cdot \beta^l$ to it, this is compatible with the way negative numbers are usually represented in computer arithmetic.[3]

As a result, the expression from line 21 is transformed from:

```
b = q + mpn_sub_n (np, np, np + n, 2 * l);
```

to

```
mpn_sub_n (np, np, np + n, 2 * l);
b = q + (bool_to_Z rb);
```

[2] In fact, the intervals $[pos_a, pos_a + l[$ and $[pos_b, pos_b + l[$ should either not overlap, or be identical, and there are similar conditions on the interval $[pos_r, pos_r + l[$.

[3] The way we handle the borrow is a bit strange. A better solution would have been to return the borrow, as is done in the C code. Our approach prevents us from describing directly algebraic expressions in which appears a call to a function. The main reason for this apparent weakness is an early problem with the CORRECTNESS tool at the time we started our experience. The C-like approach can also be implemented, but the proof development has grown quite large and changing the formalization does not seem worth the effort.

In other words, an expression containing a function returning a value in the C code is transformed according to the following principles:

- first we execute the steps involving side-effects,
- then we evaluate a side-effect-free expression.[4]

In the function's post-condition, $m@$ stands for the memory before the execution of mpn_sub_n whereas m represents the memory after the computation. The post-condition to the mpn_sub_n function as given above is not complete, as it should also express where side-effects *do not* occur. This will be seen in a later section (see §5.3).

5.2.3 Proof reuse

Most of the theorems we established at the logical description are reused in the proof of the implementation. Reusing a proof means connecting memory segments with integer variables through the interpretation function $m, p, l \mapsto \{p, l\}_m$. For instance, Theorem QleL that states $Q \leq L$ actually has the following statement:

$\forall N, L, H, H', N', N'', S', R', N_1, N_0, Q, R''.$
$H = 2H' \quad \wedge \quad 0 < L \quad \wedge \quad L \leq H \quad \wedge \quad (\mathsf{Zdivprop}\ N\ L^2\ N'\ N'') \quad \wedge$
$(\mathsf{Zdivprop}\ N''\ L\ N_1\ N_0) \quad \wedge \quad (\mathsf{Zdivprop}\ R'L + N_1\ 2S'\ Q\ R'') \quad \wedge$
$(\mathsf{SqrtremProp}\ N'\ S'\ R') \quad \wedge \quad (\mathsf{IsNormalized}\ N\ (HL)^2)$
$\Rightarrow Q \leq L.$

All values, N, N', ... appear at some point in the memory, sometimes as a combination of a memory segment and a borrow or a carry multiplied by a power of the basis. For this reason, it is possible to instantiate this theorem with some of these values, for instance in a command like the following, that appears in our formal development:

Apply QleL with
$N := \{n_{p_0}, 2n\}_{m_0}$ $L := \beta^l$ $H := \beta^h$ $N' := \{n_{p_0} + 2l, 2h\}_{m_0}$
$N'' := \{n_{p_0}, 2l\}_{m_0}$ $N_1 := \{n_{p_0} + l, l\}_{m_0}$ $N_0 := \{n_{p_0}, l\}_{m_0}$
$S' := \{s_{p_0} + l, h\}_{m_1}$ $R' := rb_1\beta^h + \{n_{p_0} + 2l, h\}_{m_1}$
$R'' := c_0\beta^h + \{n_{p_0} + l, h\}_{m_2}.$

In this example, three different states of the memory are considered. The state m_0 corresponds to the initial state of the memory when the function is called, m_1 corresponds to the memory after the recursive call to the square root function, m_2 corresponds to the state after the division of $R'L + N_1$ by $2S'$. The various premises of Theorem QleL are proven with the help of the specifications of the GMP functions, like mpn_sub_n.

[4] This approach has at least one advantage, in that it makes the order in which functions with side-effects are called explicit. It happens that the ML-like language that is taken as model in Correctness evaluates algebraic expressions in a *right-to-left* order, while the C language specifies that these expressions should be evaluated in a *left-to-right* order so that our decomposition could be meaningful.

5.3 Memory management issues

In the specification of functions that have side-effects, we must not only express that they produce an output that satisfies the intended meaning, but also that they are well-behaved: they don't attempt to access or update outside of the available memory and they leave memory outside the output segments unchanged.

5.3.1 Bounds checking

The memory has been declared as an array of size bound. This means any access in the array at an index smaller than 0 or greater than or equal to bound is illegal. This needs to be adapted when functions are supposed to access memory segments.

In our mpn_sub_n example, this leads to the following requirements:

$$pos_a + l \leq \mathsf{bound}, \qquad pos_b + l \leq \mathsf{bound}, \qquad pos_r + l \leq \mathsf{bound}.$$

Moreover pos_a, pos_b, and pos_r have been declared of type $\mathbb{N}$, so that the condition *no access under 0* is automatically satisfied. Still these conditions are added among the pre-conditions of mpn_sub_n, and similar pre-conditions are provided for our formal description of all other GMP functions.

For the mpn_dq_sqrtrem function itself, we take as pre-conditions the following inequalities:

$$n_p + 2n \leq \mathsf{bound}, \qquad s_p + n \leq \mathsf{bound}.$$

Then all the other operations operate at segments of the form $\{n_p + l, h\}$, $\{n_p + 2l, h\}$ $\{n_p + l, n\}$ or $\{s_p + l, h\}$, etc. The upper bounds for these segments are $n_p + l + h$, $n_p + 2l + h$, $n_p + l + n$, $s_p + l + h$ respectively. If we provide the information that $l + h = n$ and $l < n$, then verifying that the segments lie below bound can be performed by an automatic procedure for linear arithmetic in inequalities, like the Omega decision procedure provided in Coq. We have developed a small tactic called SolveBounds that makes sure all relevant information is provided before calling the Omega tactic. With this tactic, we have been able to get rid of all bounds checking.

5.3.2 Segments overlap

Most of GMP functions require that segments representing inputs and outputs should either be exactly the same or not overlap with each other. In our example mpn_sub_n, it means we have to add the following preconditions:

$$pos_r = pos_a \vee pos_r + l \leq pos_a \vee pos_a + l \leq pos_r$$
$$pos_r = pos_b \vee pos_r + l \leq pos_b \vee pos_b + l \leq pos_r$$
$$pos_a = pos_b \vee pos_a + l \leq pos_b \vee pos_b + l \leq pos_a$$

All these constraints can also be solved automatically by calling the above-mentioned tactic SolveBounds.

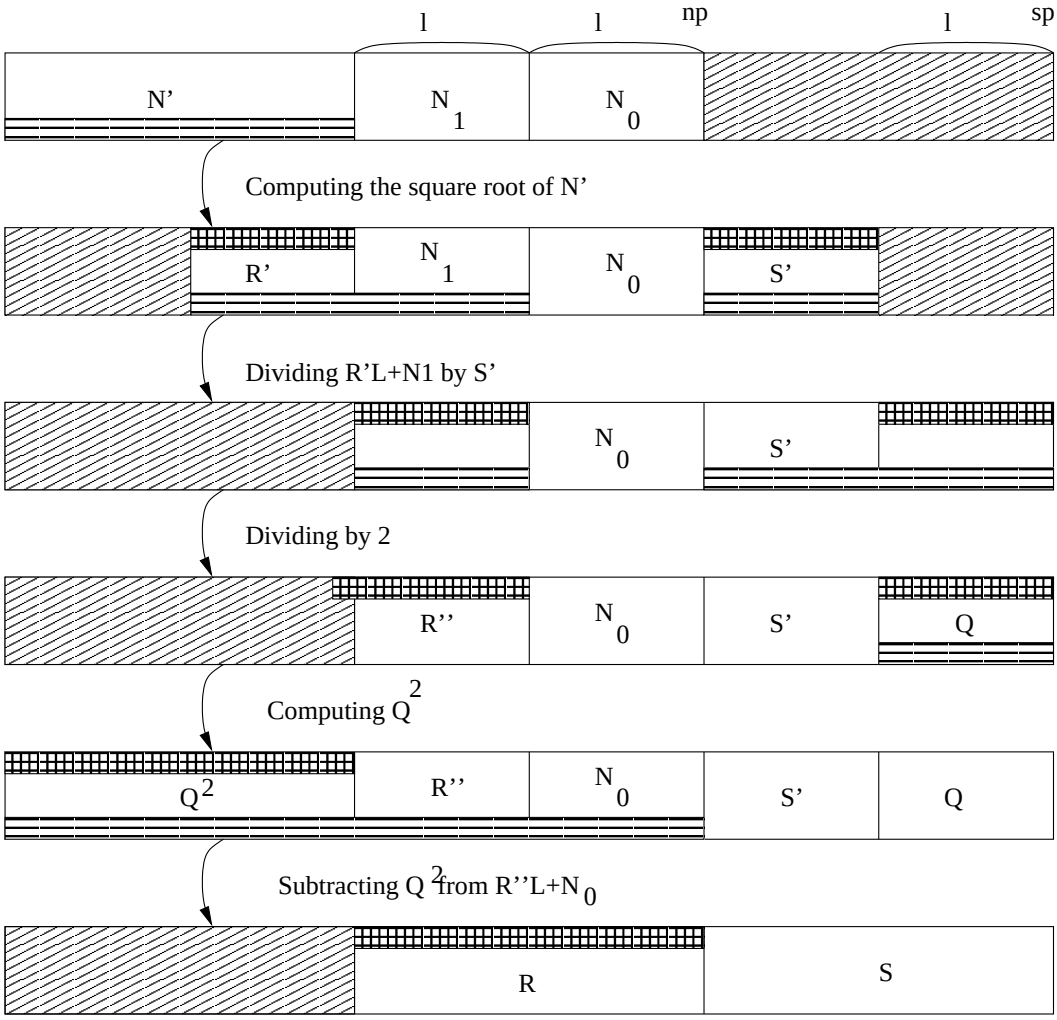

Figure 2: Memory use of the algorithm: horizontal stripes denote input for the following operation, criss-crossed patterns denote output from the previous operation, slanted stripes denote memory that is not used at that time.

5.3.3 Stability of memory areas

Another aspect of memory management is that functions leave most of the memory unchanged. The segment of memory that receives the output is modified, of course, but side-effects should be limited to that area. For instance, this property is essential to express that the value N_0 is stored unchanged in memory until the time when Q^2 is subtracted from $R''L + N_0$.

The issue of checking stability of memory areas arises in formal proofs. In hand-written proofs, this is implicit, areas for which we have no information, are considered unchanged.

For our example mpn_sub_n, we must add the following postcondition, where the notation $m[p]$ describes the value in memory at index p:

$$\forall p : \mathsf{nat}\ 0 \leq p < \mathsf{bound} \Rightarrow p < pos_r \vee pos_r + l \leq p \Rightarrow m[p] = m@[p].$$

This kind of condition makes it possible to transform a proof of equality between values in two different states of memory into a collection of comparisons. These comparisons are, once again, easy to get rid of with the help of our SolveBounds tactic. Fig. 2 gives a summary of the location and possible destruction of each value as one iteration of the recursive function executes.

5.4 Modular specification of the program

To make the proof more manageable, we actually encoded the text of the function in several sub-parts, where each sub-part was given its own set of pre- and post-conditions. This organization made it possible to share the proof work among proof developers and to limit the amount of useless facts given in the context of each proof obligation.

There are four parts, corresponding to the main phases of the computation:

- recursive call to mpn_dq_sqrtrem,
- dividing by $2S'$,
- squaring Q and subtracting it,
- correcting the result if the remainder is negative.

For the recursive call, the same phenomenon as for the functional description appears. The recursive definition is well formed if we exhibit a well-founded relation and show that some expression built with the parameters to the call decreases for this well-founded relation.

5.4.1 Dividing by $2S'$

The division by $2S'$ is performed in several steps to avoid multiplying S' by 2 and storing $2S'$ into memory. The first step consists in subtracting S' from R' if R' happens to be larger than H. This subtraction is described by the operation at line 13, but there is a trick with the borrow: The computation that is actually performed is $(R' - H) - S'$, since only the

number $R' - H$ is stored at $\{n_p + 2l, h\}$. This subtraction is bound to return a borrow, because $R' \leq 2S' < H + S'$. For this reason, the value stored at $\{n_p + 2l, h\}$ after the subtraction is:

$$R' - H - S' + H = R' - S'.$$

In this case ($R' \geq H$), it is actually the number $(R' - S')L + N_1$ that is divided by S'. This yields a quotient Q_0 and a remainder R_0 and we have the following equation:

$$(R' - S')L + N_1 = Q_0 S' + R_0,$$

which is equivalent to

$$R'L + N_1 = (L + Q_0)S' + R_0.$$

The actual division of $(R' - S')L + N_1$ by S' is done with the function mpn_divrem. This function is actually specified to perform only divisions by "normalized" numbers, with most significant bit set to 1. In this case, the numbers of limbs needed to represent the quotient can be known in advance: it is the difference between the number of dividend limbs and the number of divisor limbs. However, there may be an overflow of at most 1 bit. For our purpose this bit is added to q, in line 14. So q may be a value between 0 and 2 and, after line 14, the quotient of $R'L + N_1$ by S' is $q\beta^l + \{s_p, l\}$.

The computation on line 15 just saves the parity bit of Q_0. Lines 16 to 18 perform the actual division of the quotient by 2, and we have $0 \leq q \leq 1$ after. This division is done using bit shifts to the right, as is customary in computers using binary representation. In our formalization, we have avoided the burden of explaining why shifts could be interpreted as arithmetical operations and we have assumed that mpn_rshift performs a division by a power of 2, without further proof. For the same reason, we have replaced the operation

```
sp[l-1] |= q << (BITS_PER_MP_LIMB - 1)
```

by an addition of $\frac{\beta}{2}$ if q is odd (then it is necessarily 1), and 0 otherwise.

If the quotient Q_0 was odd, then the remainder has to be corrected by adding back S' to R_0. This correction is done on line 19. It may return an overflow that is stored in the local variable c.

5.4.2 Subtracting the square of Q

Q is represented by the number $q\beta^l + \{s_p, l\}$ at line 20, where the square of $\{s_p, l\}$ is computed. This is actually enough to compute the square of Q, because we know $Q \leq L$. From this, we can deduce that either q or $\{s_p, l\}$ is zero and the double product in the development of the square is always 0. This property is used at lines 21 and 22, where $R''L + N_0$ is represented by $c\beta^n + \{n_p, n\}$ before line 21 and Q^2 is represented by $q\beta^{2l} + \{n_p + n, 2l\}$. On line 21, the computation being performed is only $\{n_p, 2l\} - Q^2$. When $h \neq l$, then an extra subtraction of the borrow needs to be done on the most significant limb of $\{n_p, n\}$ with an update of c accordingly. As a result, the value $R''L + N_0 - Q^2$ is actually represented by $c\beta^n + \{n_p, n\}$ after line 22, where $-1 \leq c \leq 1$.

Before line 23, it is not true that $\{s_p, n\}$ contains the number $S'L + Q$. Actually, Q is represented (since line 18) by $q\beta^l + \{s_p, l\}$, so that the number $\{s_p, n\}$ actually is $S'L + Q - qL$. This is corrected on line 23.

5.4.3 Correcting the result

After line 23, the value $R''L + N_0 - Q^2$ can be found in the memory state as $c\beta^n + \{n_p, n\}$. If this value is negative, only c can make it negative, therefore testing c is enough to find out whether a correction is necessary.

The value $S'L + Q$ is represented in memory by $q\beta^n + \{n_p, n\}$, so the correction performed on the remainder in lines 26 and 27 also includes adding $2q$ to c. The subtraction in line 28 corresponds to the correction on the square root to compute $S'L + Q - 1$. The borrow returned by this subtraction is removed from q, but this is useless: q will be forgotten when the function terminates (and anyway, we already know that the ultimate value of this variable is 0).

5.5 Overview of the proof development

Improving the system. We have noticed that the tactic Omega requires more and more computation time to solve a goal when the context gets larger. To tackle this efficiency problem, we designed a new tool Gomega. It is intended to reduce the size of the context before applying a decision procedure such as Omega. The user has to give explicitly as parameters all the hypotheses required by the decision procedure to solve the goals. It is useful in the sense that it allows the user to clear the context from every assumption which is not related to memory management when proving legal array accesses for instance. Combined with a dependency analysis tool, such a tactic can be used to determine the minimum conditions that need to be verified for a statement to hold. It forces the user to say which assumptions are relevant to solve the goal.

Overall view. For the formal development, the files describing the algorithm at the logical level contain 2700 lines of formal code, the functional description contains 500 lines, and the proof of the imperative code contains 10000 lines of code. Verifying the proofs takes about ten minutes on a Pentium III 1 GHz (with 2 processors) and 1 GByte of memory. Coq uses at most 300 MBytes of memory.

6 Related work

Other researchers have worked on the formalization of arithmetic operations in theorem provers. However, most of the momentum has been provided around floating-point arithmetic and the IEEE-754 standard [Har99, Rus99, Min95, DRT01, Jac01]. The work on floating-point algorithms has more to cope with the mere correctness of the algorithms, where the intricacies of rounding errors need to be taken into account. When considering

integer arithmetic, there is little fear of misinterpreting the specifications since there is no rounding problem. As a result, integer based algorithms have been less studied. This work shows that powerful integer-based algorithms also deserve formal study. They may also be of use for the formal verification of floating-point arithmetic, as the IEEE standard suggests that floating-point number operations should be performed with the input numbers as "face values" before ultimately rounding the results.

We should also mention the work of Didier Bondyfalat [Bon02] who carried out a proof of the GMP division function used by our algorithm, although the level of abstraction is higher.

7 Conclusion

This paper describes an efficient implementation for the computation of square roots of large numbers, and its formal study using a proof assistant. The benefits of this work are a precise formal description of a program that is recognized as one of the most efficient to date, a precise description of the conditions under which this algorithm will operate correctly, and a stronger guarantee that the program will always operate as specified. We believe this is one of the first formal studies of state-of-the-art algorithms, where the formalization covers even the details of memory usage and pointer arithmetic.

The algorithm we proved is a discrete variant of Newton's method, with an inductive computation of the remainder. On the mathematical side, the major result is the fact that only one correction is needed after the division. At the implementation level, the memory usage is nearly optimal, since no additional memory is required in the recursive function, except the local variables of fixed total size which are allocated on the stack. Our work nevertheless pointed out some possible optimizations that will be studied later.

The organization of the formalization work contains three levels of abstraction. The first level of abstraction only takes care of the mathematical relationship between numbers. The second level takes care of describing the combination of basic computations, ensuring that all functions are used in their domain of applicability and that the main recursion will always terminate. The third level of abstraction gets much closer to the implementation and expresses how carries, borrows, and memory segments are handled. We believe other formalization attempts will benefit by following a similar pattern of study.

All specifications and proof developments are available on the World Wide Web (see `http://www-sop.inria.fr/lemme/AOC/SQRT/index.html`).

Acknowledgments. Thanks to the INRIA sponsored AOC project, and especially to Laurent Théry and Didier Bondyfalat. Thanks also to the Coq team, especially to Jean-Christophe Filliâtre.

References

[ABPR01] Ahmed Amerkad, Yves Bertot, Loïc Pottier, and Laurence Rideau. Mathematics and Proof Presentation in Pcoq. In *Proof Transformations, Proof Presentations and Complexity of Proofs (PTP'01)*, 2001. Sienna, Italy, also available as INRIA RR-4313.

[Bon02] Didier Bondyfalat. Certification d'un algorithme de division pour les grands entiers. Unpublished, 2002.

[Coq02] Coq development team, INRIA and LRI. *The Coq Proof Assistant Reference Manual*, January 2002. Version 7.2, available from http://coq.inria.fr/doc/main.html.

[DRT01] Marc Daumas, Laurence Rideau, and Laurent Théry. A Generic Library for Foating-Point Numbers and Its Application to Exact Computing. In *Theorem Proving in Higher Order Logics: 14th International Conference*, number 2152 in LNCS. Springer-Verlag, September 2001.

[Fil99] Jean-Christophe Filliâtre. *Preuve de programmes impératifs en théorie des types.* PhD thesis, Université Paris-Sud, July 1999.

[Fil01] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 2001. English translation of [Fil99]. To appear.

[Gra02] Torbjörn Granlund. *The GNU Multiple Precision Arithmetic Library*, 2002. Edition 4.0.1.

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In *Theorem Proving in Higher Order Logics: 12th International Conference*, number 1690 in LNCS. Springer-Verlag, September 1999.

[Jac01] Christian Jacobi. Formal verification of a theory of IEEE rounding. In Richard J. Boulton and Paul B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, 2001. Informatics Research Report EDI-INF-RR-0046, Univ. Edinburgh, UK.

[Min95] Paul S. Miner. Defining the IEEE-854 Floating-Point Standard in PVS. NASA Technical Memorandum 110167, NASA Langley Research Center, Hampton, Virginia, June 1995.

[PM93] Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993. LIP research report 92-49.

[Rus99] David M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of AMD K5 Floating Point Square-Root Microcode. *Formal Methods In System Design*, 14(1):75–125, January 1999.

[Zim99] Paul Zimmermann. Karatsuba square root. Technical Report 3805, INRIA, november 1999.

[Zur94] Dan Zuras. More on squaring and multiplying large integers. *IEEE Transactions on Computers*, 43(8):899–908, 1994.

Contents

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399