Formal Verification of a State-of-the-Art Integer Square Root

Guillaume Melquiond, Raphaël Rieu-Helft

Formal Verification of a State-of-the-Art Integer Square Root

Guillaume Melquiond*, Raphaël Rieu-Helft†*
*Inria, Université Paris-Saclay, Palaiseau F-91120
†TrustInSoft, Paris F-75014

***Abstract*—We present the automatic formal verification of a state-of-the-art algorithm from the GMP library that computes the square root of a 64-bit integer. Although it uses only integer operations, the best way to understand the program is to view it as a fixed-point arithmetic algorithm that implements Newton's method. The C code is short but intricate, involving magic constants and intentional arithmetic overflows. We have verified the algorithm using the Why3 tool and automated solvers such as Gappa.**

***Index Terms*—Formal verification, fixed-point arithmetic.**

I. Introduction

The GNU Multi-Precision library, or GMP, is a widely used, state-of-the-art library for arbitrary-precision integer arithmetic. It is implemented in assembly and C. The library is used in safety-critical contexts, but the algorithms are complex and the code is often quite intricate. Moreover, it is hard to ensure good test coverage, as some parts of the code are only used in very unlikely cases. Correctness bugs occuring with very low probability have been found in the past.[1] Formal verification of these algorithms is therefore desirable.

For square root, GMP uses a divide-and-conquer algorithm. It first computes the square root of the most significant half of its operand, and then refines this approximation using a division. The base case of the algorithm computes the square root of a 64-bit machine integer. It was written by Granlund in 2008. In this work, we focus on this base case, which is the most intricate part of the algorithm.

Indeed, although the code we are focusing on uses only integer arithmetic, it is best understood as an implementation of Newton's method using fixed-point arithmetic. To the best of our knowledge, fixed-point square root algorithms have not been the subject of formal verification work. However, there has been extensive work on floating-point square root algorithms with quadratic convergence. Harrison verified a similar algorithm for the Intel Itanium architecture using HOL Light [1]. Russinoff verified the correctness of the AMD K5 floating-point square root microcode using ACL2 [2]. Rager *et al.* used ACL2 and interval arithmetic to verify the low-level Verilog descriptions of the floating-point division and square root implementations in the SPARC ISA, and discovered new optimisations while doing so [3]. Finally, Bertot *et al.* verified the divide-and-conquer part of GMP's square root using Coq [4].

[1]Look for 'division' at https://gmplib.org/gmp5.0.html.

These verifications were all done manually using interactive proof assistants. Our proof, on the contrary, is mostly automated. We use the Why3 tool and the high-level functional language it provides, WhyML, to implement GMP's algorithms and give them a formal specification. Why3 generates verification conditions that, once proved, guarantees that the program matches its specification [5]. They cover both the mathematical correctness of the algorithm and implementation aspects such as arithmetic overflows and memory safety. We prove most of these goals automatically using SMT solvers. For goals involving fixed-point arithmetic, we use the Gappa solver [6]. Finally, we use Why3's extraction mechanism to obtain efficient, correct-by-construction C code that closely mirrors the original GMP program [7].

II. Algorithm

We give in Figure 1 the code of the 64-bit integer square root. It is essentially as found in GMP 6.1.2. For readability, some preprocessing was done, some comments and whitespace were modified, and some variables were added or renamed.

It takes an unsigned 64-bit integer `a0` larger than or equal to 2^{62} and returns its square root, storing the remainder in `*rp`. The best way to understand this algorithm is to view it as a fixed-point arithmetic algorithm, with the binary point initially placed such that the input `a0` represents $a \in [0.25; 1]$.

The main part of the algorithm consists in performing two iterations of Newton's method to approximate $a^{-1/2}$ with 32 bits of precision. As part of the last iteration, the approximation is multiplied by a to obtain a suitable approximation of $\sqrt{a}$. More precisely, we are looking for a root of $f(x) = x^{-2} - a$. Given $x_i = a^{-1/2}(1 + \varepsilon_i)$, we define $x_{i+1} = x_i - f(x_i)/f'(x_i) = x_i(3 - ax_i^2)/2$. Furthermore, if we pose ε_{i+1} such that $x_{i+1} = a^{-1/2}(1 + \varepsilon_{i+1})$, we find $|\varepsilon_{i+1}| \approx \frac{3}{2} \cdot |\varepsilon_i|^2$.

Note that the iteration can be computed with only additions, multiplications, and logical shifts. For instance, we compute x_1 as $x_0 + x_0 t_1/2$, with $t_1 \approx 1 - ax_0^2$ at line 9. The division by 2 is implicitly performed by the right shift at line 12. The absence of division primitives is the main reason why we look for an approximation of the inverse square root rather than the square root itself, which would involve a division by x_i at each step of the iteration.

The initial approximation of $a^{-1/2}$ is taken from the precomputed array `invsqrttab` of 384 constants of type `char`. Using interval arithmetic, we have checked exhaustively that

```
1 #define MAGIC 0x10000000000
2
3 mp_limb_t mpn_sqrtrem1(mp_ptr rp, mp_limb_t a0){
4   mp_limb_t a1, x0, x1, x2, c, t, t1, t2, s;
5   unsigned abits = a0 >> (64 - 1 - 8);
6   x0 = 0x100 | invsqrttab[abits - 0x80];
7   // x0 is the 1st approximation of 1/sqrt(a0)
8   a1 = a0 >> (64 - 1 - 32);
9   t1 = (mp_limb_signed_t) (0x2000000000000
10                - 0x30000 - a1 * x0 * x0) >> 16;
11  x1 = (x0 << 16) +
12       ((mp_limb_signed_t) (x0 * t1) >> (16+2));
13  // x1 is the 2nd approximation of 1/sqrt(a0)
14  t2 = x1 * (a0 >> (32-8));
15  t = t2 >> 25;
16  t = ((mp_limb_signed_t) ((a0 << 14) - t * t
17         - MAGIC) >> (32-8));
18  x2 = t2 + ((mp_limb_signed_t) (x1 * t) >> 15);
19  c = x2 >> 32;
20  // c is a full limb approximation of sqrt(a0)
21  s = c * c;
22  if (s + 2*c <= a0 - 1) {
23      s += 2*c + 1;
24      c++;
25    }
26  *rp = a0 - s;
27  return c;
28 }
```

Fig. 1. The `mpn_sqrtrem1` function.

the initial approximation `x0` has a relative error ε_0 smaller than about $2^{-8.5}$ for all values of `a0`. It follows that after an iteration, we have $|\varepsilon_1| \lesssim 2^{-16.5}$ and after two steps, $|\varepsilon_2| \lesssim 2^{-32.5}$. The square root of `a0` can be represented using at most 32 bits, so we would expect the final approximation to be either exactly the truncated square root of `a0` or off by one. Note that we are computing using fixed-point numbers rather than real numbers, so additional rounding errors are introduced during the computation and worsen this error bound somewhat. However, the quadratic nature of the convergence absorbs most of the rounding errors from the first iteration, and the final result is still off by at most one. The magic constants `0x30000` and `MAGIC`, which would not be part of an ideal Newton iteration, ensure that the approximation is always by default. As a result, the approximation after two steps is known to be either exactly the truncated square root, or its immediate predecessor. The final `if` block performs the required adjustment step, by adding 1 to the result if it does not make its square exceed `a0`.

There are several implementation tricks that make this algorithm both efficient and difficult to prove. For instance, some computations intentionally overflow, such as the left shift and the multiplication at line 16. In this case, `t` is represented using 39 bits, and `a0` uses all 64 bits, so both computations overflow by 14 bits. However, the top part of `t*t` is known to be equal to the top part of `a0`, and these numbers are subtracted, so no information is lost. Another thing to note is that the variable `t`, which is an error term, essentially represents a signed value even though it is an unsigned 64-bit integer. Indeed, we cast it into a signed integer before shifting it to the right. This means that its sign is preserved by the shift. However, `t` could not be represented as an actual signed integer because some computations involving it overflow, such as the product `t*t` at line 16.

III. Modeling

For the most part, the C code uses only one number type: unsigned 64-bit integers. Relevant arithmetic operations are addition, multiplication, left shift, and right shift. We could write the WhyML code using them, but we would lose some helpful information about the algorithm, as these integers are fixed-point representations of real numbers. So, it is important to carry around the position of their binary point. To do so, we introduce a new WhyML record type with two fields:

```
type fxp = { ival: uint64; ghost iexp: int }
```

Notice the `ghost` keyword. It specifies that the `iexp` field, which denotes the position of the binary point, is only there for the sake of the proof. No actual code will ever look at its value. So, from an implementation point of view, a value of type `fxp` is indistinguishable from the value of its field `ival`, which is an unsigned 64-bit integer. Notice also that the type of the `iexp` field is `int`, which is the type of unbounded integers. Indeed, since the field will be invisible from the code, we might just as well choose the type that makes the verification the simplest. In fact, most of the verification will be performed using unbounded integers and real numbers.

Now that we have the `fxp` type, we can specify the basic arithmetic operations on fixed-point values. As an illustration, here is part of the declaration of addition. It takes two fixed-point numbers x and y as inputs and returns a fixed-point number denoted *result* in the specification. The precondition requires that the binary points of x and y are aligned; the user will have to prove this property. The postcondition ensures that the result is aligned with x (and thus y too); this property will be available in any subsequent proof.

```
val fxp_add (x y: fxp): fxp
  requires { iexp x = iexp y }
  ensures  { iexp result = iexp x }
  ensures  { rval result = rval x +. rval y }
```

The last line of the specification ensures that the real number represented by *result* is the sum of the two real numbers represented by x and y. Notice that, to distinguish operators on real numbers from operators on unbounded integers, they are suffixed with a point, *e.g.*, "`+.`", "`*.`". Moreover, the `rval` function, which turns a fixed-point value into the real number it represents, can often be inferred by Why3. So, this clause could have been written just "`result = x +. y`". In the following, most occurrences of `rval` will thus be implicit, for the sake of readability.

We have not said anything about the definition of `rval` yet. We could simply define the real number as $rval = ival \cdot 2^{iexp}$. Unfortunately, that would make the algorithm impossible to prove. Indeed, as explained before, several 64-bit operations

might overflow and cause their results to wrap around, while this is not the case of the real numbers they are meant to represent. To circumvent this issue, we add `rval` as another field of `fxp`. We also add two type invariants to state that the values of the three fields are related. Now, whenever the code creates a fixed-point value, the user has to prove that the invariants hold.

```
type fxp = { ival: uint64; ghost rval: real;
                           ghost iexp: int }
  invariant { rval = floor_at rval iexp }
  invariant { ival =
    mod (floor (rval *. pow2 (-iexp)))
        (uint64'maxInt + 1) }
```

The first invariant forces the real number to be a multiple of 2^{iexp} by stating that *rval* is left unchanged by truncating it at this position. The second invariant states that the 64-bit integer can be obtained by first scaling the real number so that it becomes an integer, and then making it fit into the `uint64` type using the remainder of an Euclidean division. The `floor_at` function that appears in the first invariant is defined as follows:

```
function floor_at (x: real) (k: int): real =
  floor (x *. pow2 (-k)) *. pow2 k
```

A fixed-point value can be created using the following function. It takes a 64-bit integer, usually a literal one, and the position of the binary point. Since this position is ghost, and so are the fields `iexp` and `rval`, this function is effectively the identity function.

```
let fxp_init (x: uint64) (ghost k: int): fxp
= { ival = x; iexp = k; rval = x *. pow2 k }
```

Subtraction and multiplication are not fundamentally different from addition. Left and right shifts are more interesting. Contrarily to plain integer arithmetic, their role is not just to perform some cheap multiplication or division by a power of two; they can also be used to move the binary point to a given position. Let us illustrate this situation with the following function, which performs an arithmetic right shift "`(mp_limb_signed_t)x >> k`".

```
val fxp_asr' (x: fxp) (k: uint64)
             (ghost m: int): fxp
  requires { int64'minInt *. pow2 (iexp x) <=.
    rval x <=. int64'maxInt *. pow2 (iexp x) }
  ensures { iexp result = iexp x + k - m }
  ensures { rval result =
    floor_at (rval x *. pow2 (-m))
             (iexp x + k - m) }
```

The precondition requires that the real number represented by x is bounded, so that the sign bit contains enough information to fill the $k+1$ most significant bits. In particular, to use this function, the user will have to prove that, if some previous operation overflowed, it has been compensated in some way. The ghost argument m tells how much of the shift is actually a division of the real number, so the binary point is moved by $k-m$ as stated by the first postcondition. (The `fxp_asr` variant corresponds to $m=0$.) Finally, the second postcondition expresses that the real result is $x \cdot 2^{-m}$ except for the least significant bits that are lost.

This loss of accuracy is expressed using the aforementioned `floor_at` function. This function can be interpreted as a rounding toward $-\infty$ in some fixed-point format. As such, when verification conditions will be sent by Why3 to Gappa, an expression "`floor_at x p`" will be translated to "`fixed<p,dn>(x)`". Note that Gappa requires `p` to be an integer literal, so it would choke on "`iexp x + k - m`". So, we improved Why3 a bit to make it precompute such expressions before sending them to Gappa. As a side effect, this change also made it possible to turn constant expressions such as "`int64'minInt *. pow2 (iexp x)`" into proper numbers. That is the only change we had to make to Why3.

IV. Proof

Once fixed-point arithmetic has been modeled as a Why3 theory, translating GMP's square root algorithm from C code to WhyML is mostly straightforward. The only details we had to guess are the ghost arguments passed to functions `fxp_init` and `fxp_asr'`, which are obvious only if one knows that the algorithm implements Newton's method. The complete specification and part of the WhyML code are given in Figure 2. The specification requires that the function is called with a valid pointer `rp` and an integer `a0` large enough. It ensures that the unsigned integer returned by the function is the truncated square root and that the unsigned integer pointed by `rp` holds the remainder. Assertions and "`let ghost`" lines are only here to help state intermediate assertions; they do not carry any computational content.

Let us describe the verification process now. The most critical part of the algorithm is the verification of the following assertion, which states that the fixed-point value x approximates $sa = 2^{-32}\lfloor\sqrt{a_0}\rfloor$ with an accuracy of 32 bits:

```
assert { -0x1.p-32 <=. x -. sa <=. 0. };
```

This is exactly the kind of verification condition Gappa is meant to verify. Unfortunately, Gappa is unable to automatically discharge it, as it does not know anything about Newton's method. So, we have to write additional assertions in the code to help Gappa. In particular, we have to make it clear that the convergence is quadratic. For example, the following equality gives the relative error between $a^{-1/2}$ and an idealized version x_1' of the second approximation `x1`.

$$\frac{x_1' - a^{-1/2}}{a^{-1/2}} = -\tfrac{1}{2}(\varepsilon_0^2(3+\varepsilon_0)+\ldots).$$

The value x_1', denoted by `mx1` in Figure 2, is not exactly the theoretical x_1 from the Newton iteration, as it replaces a by `a1` and takes into account the magic constant `0x30000`. However, the formula does not contain any rounding operator, so it can be obtained using a computer algebra system without much trouble. From this formula, Gappa will then deduce a way to bound the relative error between $a^{-1/2}$ and `x1`, despite the rounding errors.

All in all, Gappa needs four equalities to be able to prove the main error bound. Two of them are the relative errors

```
let sqrt1 (rp: ptr uint64) (a0: uint64): uint64
  requires { valid rp 1 }
  requires { 0x4000000000000000 <= a0 }
  ensures { result*result <= a0
            < (result+1)*(result+1) }
  ensures { result*result + get rp = a0 }
=
  let a = fxp_init a0 (-64) in
  assert {0.25 <=.a<=. 0xffffffffffffffffp-64};
  assert { 0. <. a };
  let ghost rsa = 1. /. sqrt a in
  let x0 = rsa_estimate a in
  let ghost e0 = (x0 -. rsa) /. rsa in
  let a1 = fxp_lsr a 31 in
  let ghost ea1 = (a1 -. a) /. a in
  let m1 = fxp_sub
    (fxp_init 0x2000000000000 (-49))
    (fxp_init 0x30000 (-49)) in
  let t1' = fxp_sub m1
    (fxp_mul (fxp_mul x0 x0) a1) in
  let t1 = fxp_asr t1' 16 in
  let x1 = fxp_add (fxp_lsl x0 16)
     (fxp_asr' (fxp_mul x0 t1) 18 1) in
  let ghost mx1 = x0 +. x0 *. t1' *. 0.5 in
  assert { (mx1 -. rsa) /. rsa =
    -0.5 *. (e0*.e0 *. (3.+.e0) +. (1.+.e0) *.
      (1.-.m1 +. (1.+.e0)*.(1.+.e0) *. ea1)) };
```

Fig. 2. The WhyML function up to the assertion on the first Newton iteration.

of both Newton iterations. The other two are just there to help Gappa fight the dependency effect of interval arithmetic, *e.g.*, $a \cdot a^{-1/2} = \sqrt{a}$. All these equalities are written as WhyML assertions and have to be discharged by Why3 using some other prover. Fortunately, the `field` tactic makes it straightforward to do so using the Coq proof assistant, once it has been told $a = \sqrt{a}^2$.

The verification went surprisingly well and Gappa was able to discharge all the preconditions of the right-shift functions. It failed at discharging the main error bound, though. (It would have presumably succeeded, had we let it run long enough.) We first tried to modify slightly the equality describing the relative error of the second Newton iteration. This made the verification faster, but it would still have taken hours at best. So, we ended up modifying the code a bit. GMP's square root makes use of the magic constant 2^{40}; we replaced it by $2.25 \cdot 2^{40}$. Gappa was then able to discharge the verification condition in a few seconds.

This does not mean that GMP's algorithm is incorrect. It just means that Gappa overestimates some error, which prevents it from proving the code for the original constant. Interestingly enough, a comment in the C code indicates that this magic constant can be chosen between $0.9997 \cdot 2^{40}$ and $35.16 \cdot 2^{40}$.

Once we have proved $x - sa \in [-2^{-32}; 0]$, proving $c - \lfloor\sqrt{a}\rfloor \in [-1; 0]$ (with c the integer representing x) is mostly a matter of unfolding the definitions and performing some simplifications. Then, by stating a suitable assertion, we get SMT solvers to automatically prove that none of the operations of $c \cdot c + 2c$ can overflow. All that is left is verifying the postcondition of the function, which SMT solvers have no difficulty discharging automatically.

V. Conclusion

In the end, we have a fully verified WhyML implementation of a Newton-based fixed-point algorithm, GMP's one-limb square root.[2] The only difference lies in the choice of the second magic constant, to avoid an inefficiency in Gappa, presumably. For the sake of completeness, the constant seeds, which are hidden behind the `rsa_estimate` call, should also be verified using Why3 rather than using external tools, once WhyML supports literal arrays. The C code generated by Why3 for the square root is identical to GMP's code, assuming some simple variable propagation that any optimizing compiler is able to do. In particular, the resulting C code is binary-compatible and could be transparently substituted to the one from GMP.

Writing the fixed-point arithmetic theory, understanding GMP's algorithm, translating it to WhyML, and verifying it, were the matter of a few days. The process was much shorter than we initially expected, as we did not have to give that many hints for Gappa to discharge the most intricate verification conditions. Moreover, the ability to write ghost code in WhyML made it possible to interleave these hints with the code. As a consequence, the assertions not only act as intermediate proof steps, they also serve to document why the code works. As for the verification conditions that are outside the scope of Gappa, they did not cause any major trouble and were discharged either automatically using SMT and TPTP solvers or interactively using some short Coq scripts. The total execution time of the provers (including Gappa) was under 30 seconds. We have also proved the divide-and-conquer case of the square root algorithm. Thus, square root is now part of our verified C library of GMP algorithms, which already included addition, multiplication, logical shifts, and division.

References

[1] J. Harrison, "Formal verification of square root algorithms," *Formal Methods in System Design*, vol. 22, no. 2, pp. 143–153, 2003.

[2] D. M. Russinoff, "A mechanically checked proof of correctness of the AMD K5 floating point square root microcode," *Formal Methods in System Design*, vol. 14, no. 1, pp. 75–125, 1999.

[3] D. L. Rager, J. Ebergen, D. Nadezhin, A. Lee, C. K. Chau, and B. Selfridge, "Formal verification of division and square root implementations, an Oracle report," in *16th Conference on Formal Methods in Computer-Aided Design*, Oct. 2016, pp. 149–152.

[4] Y. Bertot, N. Magaud, and P. Zimmermann, "A proof of GMP square root," *Journal of Automated Reasoning*, vol. 29, no. 3-4, pp. 225–252, 2002.

[5] J.-C. Filliâtre and A. Paskevich, "Why3 — where programs meet provers," in *22nd European Symposium on Programming*, ser. LNCS, vol. 7792, Mar. 2013, pp. 125–128.

[6] M. Daumas and G. Melquiond, "Certification of bounds on expressions involving rounded operators," *Transactions on Mathematical Software*, vol. 37, no. 1, pp. 1–20, 2010.

[7] R. Rieu-Helft, C. Marché, and G. Melquiond, "How to get an efficient yet verified arbitrary-precision integer library," in *9th Working Conference on Verified Software: Theories, Tools, and Experiments*, ser. LNCS, vol. 10712, Heidelberg, Germany, Jul. 2017, pp. 84–101.

[2] The library can be found at https://www.lri.fr/~rieu/mp.html.