How to Get an Efficient yet Verified Arbitrary-Precision Integer Library

Raphaël Rieu-Helft, Claude Marché, Guillaume Melquiond

How to Get an Efficient yet Verified Arbitrary-Precision Integer Library

Raphaël Rieu-Helft[1,2,3], Claude Marché[2,3], and Guillaume Melquiond[2,3]

[1] École Normale Supérieure, F-75230 Paris
[2] Inria, Université Paris-Saclay, F-91120 Palaiseau
[3] LRI (CNRS & Univ. Paris-Sud), Université Paris-Saclay, F-91405 Orsay

Abstract. The GNU Multi-Precision library is a widely used, safety-critical, library for arbitrary-precision arithmetic. Its source code is written in C and assembly, and includes intricate state-of-the-art algorithms for the sake of high performance. Formally verifying the functional behavior of such highly optimized code, not designed with verification in mind, is challenging. We present a fully verified library designed using the Why3 program verifier. The use of a dedicated memory model makes it possible to have the Why3 code be very similar to the original GMP code. This library is extracted to C and is compatible and performance-competitive with GMP.

Keywords: arbitrary-precision arithmetic, deductive program verification, C language, Why3 program verifier

1 Introduction

The GNU Multi-Precision library,[4] GMP for short, is a widely used library for arithmetic on integers and rational numbers of arbitrary size. Its applications range from academic research (e.g. research on computational algebra) to concrete applications of our daily life (e.g. security of Internet applications). Some of these applications make GMP safety-critical (e.g. cryptographic primitives). In this paper we focus on the `mpn` component of GMP, which is dedicated to non-negative integers and is used as a basis in all others components. For maximal performance, GMP uses numerous state-of-the-art algorithms for basic operations like addition, multiplication, and division; these algorithms are selected depending on size of the numbers involved. Moreover, the implementation is written in low-level C code, and depending on the target computer architecture, some parts are even rewritten in assembly.

Being highly optimized for run-time efficiency, the code of GMP is intricate and thus error-prone. It is extensively tested but it is hard to reach a satisfactory coverage in practice: the number of possible inputs is very large, the different branches of the algorithms are numerous, and some of them are taken with a very

[4] `http://gmplib.org/`

low probability (some branches are taken with probability 2^{-64} or less). Verifying the code for all inputs using static program verification is thus desirable. Such a verification, however, is difficult, not only because of the intrinsic complexity of the algorithms, but also because the code is written in a low-level language with performance in mind, but not verification. In this paper we present an approach to address this latter challenge.

The main idea of our approach is to first write the code in some higher-level language, namely the programming language WhyML supported by the Why3 verification environment. This language is designed for static verification with respect to some functional behavior specified using an expressive formal specification language. The main issue is then to convert such a high-level code into an efficient executable code. Our approach is to first design a dedicated *memory model* in Why3, on top of which we then implement our functions. This memory model is designed to permit a direct compilation from WhyML to C. As a result, we obtain the first fully verified library, compatible with GMP (function signatures are the same), and almost as efficient as GMP on medium-sized integers (up to around 20 words of 64 bits). The full development is available from `http://www.lri.fr/~rieu/mp.html`.

The paper is organized as follows. In Section 2, we present the design of our dedicated memory model and explain how it is suitable for compilation to C. In Section 3, we present the specifications and the algorithms we implemented for arithmetic operations including division which is the trickiest. In Section 4, we present an extensive evaluation of the efficiency of the generated code, comparing it with GMP. We discuss related work in Section 5 and we conclude in Section 6.

2 From WhyML to C

Why3 is an environment for deductive program verification, providing a rich language for specification and programming, called WhyML. WhyML is used as an intermediate language for verification of C, Java, and Ada programs [10,15], and is also intended to be comfortable as a primary programming language [11]. WhyML function definitions are annotated with pre- and postconditions both for normal and exceptional termination, and loops are annotated with invariants.

The specification component of WhyML [4,7], used to write program annotations and background theories, is an extension of first-order logic. It features ML-style polymorphic types (prenex polymorphism), algebraic data types, inductive and co-inductive predicates, recursive definitions over algebraic types. Constructions like pattern matching, let-binding, and conditionals, can be used directly inside formulas and terms. Why3 comes with a rich standard library providing general-purpose theories useful for specifying programs, including integer and real arithmetic. From programs annotated with specifications, Why3 generates proof obligations and dispatches them to multiple provers, including SMT solvers Alt-Ergo, CVC4, Z3; TPTP first-order provers E, SPASS, Vampire; interactive theorem provers Coq, Isabelle, and PVS. As most of the provers do not support some of the language features, typically pattern matching, polymor-

```
type int32
val function to_int (n:int32) : int
meta coercion function to_int
predicate in_bounds (n:int) = - 0x8000_000 ≤ n ≤ 0x7fff_ffff
axiom to_int_in_bounds: forall n:int32. in_bounds (to_int n)
val mul (x y:int32) : int32
  requires { in_bounds (to_int x * to_int y)  }
  ensures  { to_int result = to_int x * to_int y }
```

Fig. 1: Excerpt from the specification of 32-bit machine words in Why3.

phic types, or recursion, Why3 applies a series of encoding transformations to eliminate unsupported constructions before dispatching a proof obligation.

The programming part of WhyML is a dialect of ML with a number of restrictions to make automated proving easier. The major restriction concerns the potential aliasing of mutable data structures. The language and its typing system are designed so that all aliases are statically known. Technically, the typing system computes *read* and *write effects* on *singleton regions* for each sub-expressions [8]. These effects allow the design of a weakest precondition calculus that is as simple as for the *while* languages usually considered in classical Hoare logic. Verification of complex code with Why3 and automatic provers typically expects user guidance through addition of intermediate assertions [16] and verification-only code (*ghost code*) [9]. See Why3's Web site[5] for an extensive tutorial and a large collection of examples [5].

The *extraction* mechanism of Why3 amounts to compiling WhyML code into a regular programming language while forgetting verification-only annotations. Why3 natively supports extraction to OCaml. For our work we had to implement extraction to C code. To obtain C code that includes low-level memory access through pointers, it was mandatory to start by designing a Why3 model of the C memory heap and pointers, where potential pointer aliasing is controlled in a way that accommodates WhyML typing system. The description of this memory model and the extraction to C is the purpose of the rest of this section.

2.1 Machine words and arithmetic primitives

In WhyML, only the type `int` of unbounded mathematical integers is a built-in data type. Machine integers are defined instead in Why3 standard library, specified either in term of intervals of mathematical integers or with bitvectors [13]. We use the first option here, which is roughly described in Fig. 1 for signed 32-bits words. The type `int32` is abstract, equipped with a projection `to_int` mapping words to their mathematical value. Predicate `in_bounds` together with axiom `to_int_in_bounds` specify their possible range. Arithmetic operators like multiplication are then specified in terms of a pre-condition preventing overflows,

[5] `http://why3.lri.fr/`

```
let constant max = 0xffff_ffff_ffff_ffff
val mul_mod (x y:uint64) : uint64
  ensures { to_int result = mod (x * y) (max+1) }
val mul_double (x y:uint64) : (uint64,uint64)
  returns { (l,h) → l + (max+1) * h = x * y }
```

Fig. 2: Multiplication operations on `uint64`.

and a post-condition giving the expected value of the result. Notice the special `meta` declaration which is a recent addition in Why3. It indicates that `int32` words should be implicitly cast to their integer values in specifications. For example, in the contract of function `mul`, we could omit all occurrences of `to_int`, something that we do in the rest of the paper.

To implement arbitrary-precision arithmetic, we had to add primitive operations that allow overflows. This is shown in Fig. 2 for unsigned 64-bit words. The function `mul_mod` has a wrap-around semantics (result is taken modulo 2^{64}), while the function `mul_double` returns the full product as a pair of words. Similarly, addition and subtraction come in different flavors (defensive against overflow, 2-complement, with carry in/out). Logical shifts also have both a defensive version and a version with a two-word output. Finally, there is only one division primitive, which takes a two-word numerator and a one-word denominator, and computes a quotient and a remainder.

Regarding extraction, all these data types for machine words are translated into their relevant C types (e.g. `uint64_t`). The axiomatized operations are replaced by their equivalent native C functions when possible. For example, both operations `mul` and `mul_mod` are extracted to C multiplication, since C operators on unsigned integer types are guaranteed to have the expected semantics for overflows. The `mul_double` operation, however, does not map to any C operator, so we import the corresponding operation from GMP's `longlong.h` file. Reusing GMP's primitives does not only make our library portable to numerous architectures, but it also makes for fairer benchmarks, allowing us to compare the efficiency of big integer algorithms independently of the primitives.

2.2 A simple model for C pointers and heap memory

Arbitrary-size integers are represented in C as buffers of unsigned machine words. The functions manipulate pointers, make use of aliasing, and sometimes operate in place. To implement these functions in WhyML, we design a model where the needed pointer operations are axiomatized, as shown in Fig. 3. At extraction, these operations are then directly replaced by their C equivalents, indicated as comments in Fig. 3. Our model only specifies the C features we need. For pointer arithmetic, we only model incrementation of a pointer by an integer, as we have no use for pointer comparisons or subtractions. We do not need pointer cast either, nor do we need the C address-of operator `&`.

```
1  type ptr 'a = { mutable data : array 'a ; offset : int }
2
3  function plength (p:ptr 'a) : int32 = p.data.length
4
5  function pelts (p:ptr 'a) : (int → 'a) = p.data.elts
6
7  val malloc (sz:uint32) : ptr 'a  (*   malloc(sz * sizeof('a))  *)
8    requires { sz > 0 }
9    ensures  { plength result = sz ∨ plength result = 0 }
10   ensures  { result.offset = 0 }
11
12 val free (p:ptr 'a) : unit                        (*   free(p)  *)
13   requires { p.offset = 0 }
14   writes   { p.data }
15   ensures  { plength p = 0 }
16
17 predicate valid (p:ptr 'a) (sz:int) =
18   0 ≤ sz ∧ 0 ≤ p.offset ∧ p.offset + sz ≤ plength p
19
20 val get (p:ptr 'a) : 'a                            (*     *p   *)
21   requires { 0 ≤ p.offset < plength p }
22   ensures  { result = p.data[p.offset] }
23
24 val set (p:ptr 'a) (v:'a) : unit                   (*   *p = v  *)
25   requires { 0 ≤ p.offset < plength p }
26   writes   { p.data.elts }
27   ensures  { pelts p = Map.set (pelts (old p)) p.offset v }
28
29 val incr (p:ptr 'a) (ofs:int32) : ptr 'a           (*   p+ofs  *)
30   requires { p.offset + ofs ≤ plength p }
31   alias    { p.data ~ result.data }
32   ensures  { result.offset = p.offset + ofs }
33   ensures  { result.data = p.data }
34
35 val get_ofs (p:ptr 'a) (ofs:int32) : 'a         (*   *(p+ofs)  *)
36   requires { 0 ≤ p.offset + ofs < plength p }
37   ensures  { result = p.data[p.offset + ofs] }
```

Fig. 3: A Why3 memory model for C pointers and heap memory.

The C heap memory is seen as a set of memory blocks called *objects* in the C99 standard. The WhyML polymorphic type `ptr 'a` (Fig. 3, line 1) represents pointers to blocks storing data of type `'a`. The field `data` of a pointer is an array containing the block content, while the field `offset` indicates which array cell it points to. This construction supports pointer aliasing: several pointers may reference the same array (and thus point inside the same memory block),

and thanks to WhyML region-based type system, an assignment through one pointer is propagated to other pointers.

Pointers are allocated by the `malloc` function. In case of failure it returns an *invalid* pointer, represented by a block of length 0. As such, we forbid passing 0 to `malloc`. The `free` function invalidates its parameter by setting the length of its block to 0. A pointer is considered *valid for a size s* (Fig. 3, line 17) if its offset plus *s* does not exceed the size of its block. The function `get` (line 20) represents pointer dereferencing for reading. The function `set` represents memory assignment; the `writes` clause specifies the expected write effect on the block.

The `incr` function (line 29) returns the sum of a pointer and an integer. Just as in the C standard [14, Section 6.5.6, "Additive Operators"], one may only compute a pointer that points inside a valid block or to the element just past it. The Why3 keyword `alias` in the signature of `incr` declares the aliasing of the returned pointer with the pointer parameter. Behind the scenes, it unifies the regions of `p.data` and `result.data` [8]. This aliasing is correct not only with respect to setting the contents of the pointed block, but also with respect to `free`. This makes it possible to write a particularly short specification for `free`: the `writes` effect on `p.data` induces a so-called *reset* on it [8], meaning that the region formerly pointed by `p` can no longer be accessed by any of its aliases, which are invalidated.

2.3 Extracting to idiomatic C code

The main objective of our extraction is to produce code that is correct and as efficient as possible for our arbitrary-precision library. Some WhyML language features, such as algebraic types and higher-order functions, are hard to translate into C because they would require introducing complex constructions like closures and automatic memory allocation and deallocation. Therefore, we decided to support only a small fragment of the WhyML language in our extraction. The goal is not so much to extract arbitrary WhyML code to C as to extract imperative, almost C-like WhyML code to a simple and efficient C program. The supported features of WhyML are those that can be translated straightforwardly to C, such as loops or references. What we gain by giving up on so many language features is that the extraction process is extremely straightforward, and the extracted code resembles the WhyML code line-to-line, with very little added complexity. This makes it easier to obtain efficient C code, as the WhyML programmer can have a good idea of what the extracted code will be like. The straightforwardness of the extraction also gives a measure of additional trust in the extracted code and in the extraction process, which is not formally verified.

We now present in more details a few language features that we need to design our library, for which the translation to C is not direct.

Compilation of exceptions into `break` or `return` statements. WhyML does not support certain standard imperative constructs natively. For example, there is neither `break` nor `return` constructs in WhyML. Yet some GMP algorithms, such as big integer comparison (Section 3.2), make use of these features.

```
let f (a:int32) : (int32,int32) =
  let b = a+a in
  (a,b)

let g () : int32 =
  let x = Int32.of_int 42 in
  let (y,z) = f x in
  z - y
```

```
void f(int32_t * result,
       int32_t * result1,
       int32_t a) {
  int32_t b;
  b = (a + a);
  (*result) = a;
  (*result1) = b;
  return;
}

int32_t g() {
  int32_t y, z;
  f(&y, &z, 42);
  return (z - y);
}
```

Fig. 4: WhyML function returning a tuple (on the left) and its C extraction.

To encode them in WhyML, we use its ML-style exception mechanism. Our extraction recognizes when these exceptions can be turned into `break` or `return` statements. For `break`, we essentially detect the following pattern and extract all instances of `raise B` in the body of the loop (but not inside potential inner nested loops) as `break`.

```
try while ... do ... raise B ... done with B → () end
```

For `return`, we similarly detect the following pattern of function definitions and extract all instances of `raise (R e)` as `return e`.

```
let f (args) = ... ; try ... raise (R e) ... with R v → v end
```

Note that the `try with` construct must be in tail position of the function body. Our extraction recognizes these patterns independently of the names of the exceptions being used. Any `try with` or `raise` construct that does not fit in any of these patterns causes the program to be rejected by our extraction.

Multiple return values. Many of our WhyML functions, particularly arithmetic primitives, return multiple values in a tuple. For example, this is the case of the `mul_double` primitive (Fig. 2). This has no native equivalent in C. We choose to extract each function returning a tuple as a C function returning `void`, taking as extra parameters a pointer per component of the tuple. We detect the call pattern

```
let (x1,x2,...) = f(args) in ...
```

and extract it as

```
f(&x1,&x2,..., args); ...
```

Fig. 4 is an example of WhyML code that defines and calls a function that returns a tuple and the corresponding generated C code.

3 Computing with arbitrary-precision integers

3.1 Algorithm specifications

Just as in GMP, we represent natural integers as buffers of unsigned integers called *limbs*. We set a radix β (generally $\beta = 2^{32}$ or 2^{64}, but the proofs only require it to be a power of 2). Any natural number N has a unique radix-β decomposition $\sum_{k=0}^{n-1} a_k \beta^k$, which is represented as the buffer $a_0 a_1 \dots a_{n-1}$ (with the least significant limb first).

For efficiency, there is no memory management in the low-level functions, so the caller code has to keep track of number sizes. Operands are specified by a pointer to their least significant limb and a limb count of type `int32`.

```
type limb = uint64
type t = ptr limb
```

If a pointer a is valid over a size n, we denote:

$$\texttt{value}(a, n) = \overline{a_0 \dots a_{n-1}} = \sum_{k=0}^{n-1} a_k \beta^k.$$

In our Why3 development, `value` is defined recursively

```
let rec ghost function value_sub (x:map int limb) (n:int) (m:int) : int
  variant {m - n}
= if n < m then x[n] + radix * value_sub x (n+1) m else 0

function value (x:t) (sz:int) : int
  = value_sub (pelts x) x.offset (x.offset + sz)
```

While the functions of our library use only machine types (pointers, limbs, etc), their specifications are expressed in terms of mathematical integers through extensive use of the function `value`. As an example, Fig. 5 shows the specification of the addition function. Note that the region-based type system forbids aliasing `r` with `x` or `y`. Notice also that the specification is well-typed because the conversion functions from `int32` and `limb` to `int` are coercions: otherwise many applications of `to_int` would be required.

3.2 Example of proved algorithm: comparison

Let us look at the Why3 implementation of GMP's `mpn_cmp` function, shown in Fig. 6. Just like GMP, this is the only comparison function on natural integers provided by our library. The `mpn_cmp` function takes two pointers to the integers as arguments, as well as the size of the pointed buffers. It returns -1, 0, 1, depending on the way the numbers are ordered. Our implementation has the same interface and the same behavior. The algorithm is very straightforward: it simply iterates both operands until it finds a difference, starting at the most significant limb. Once a difference is found, we can conclude immediately. If no difference is found, then the integers are equal.

```
(** [wmpn_add r x sx y sy] adds [(x, sx)] to [(y,sy)] and writes the
result in [(r, sx)].  [sx] must be greater than or equal to
[sy]. Returns carry, either 0 or 1. Corresponds to [mpn_add]. *)
let wmpn_add (r:t) (x:t) (sx:int32) (y:t) (sy:int32) : limb
  requires { 0 ≤ sy ≤ sx }
  requires { valid x sx }
  requires { valid y sy }
  requires { valid r sx }
  writes   { r.data.elts }
  ensures  { 0 ≤ result ≤ 1 }
  returns  { carry → value r sx + (power radix sx) * carry
                   = value x sx + value y sy }
```

Fig. 5: Specification of `wmpn_add`.

```
1  let wmpn_cmp (x y:t) (sz:int32) : int32
2    requires { valid x sz }
3    requires { valid y sz }
4    ensures  { result = compare_int (value x sz) (value y sz) }
5  = let i = ref sz in
6    try
7      while Int32.(≥) !i (Int32.of_int 1) do
8        variant { to_int !i }
9        invariant { 0 ≤ !i ≤ sz }
10       invariant { forall j. !i ≤ j < sz →
11                   (pelts x)[x.offset+j] = (pelts y)[y.offset+j] }
12       i := Int32.(-) !i (Int32.of_int 1);
13       let lx = get_ofs x !i in let ly = get_ofs y !i in
14       if (Limb.ne lx ly) then
15         if Limb.(>) lx ly
16         then raise (Return32 (Int32.of_int 1))
17         else raise (Return32 (Int32.of_int (-1)))
18       end
19     done;
20     Int32.of_int 0
21   with Return32 r → r
22   end
```

Fig. 6: Why3 implementation of `mpn_cmp`.

The most important part of the proof is the loop invariant at line 10: both source operands are identical from offsets $i+1$ to n. The following lemma is used to prove the postcondition. It simply states that two big integers have the same value if their limbs are equal.

```
int32_t wmpn_cmp(uint64_t * x, uint64_t * y, int32_t sz) {
  int32_t i, o;
  uint64_t lx, ly;
  i = (sz);
  while (i >= 1) {
    o = (i - 1); i = o;
    lx = (*(x+(i)));
    ly = (*(y+(i)));
    if (lx != ly) {
      if (lx > ly) return (1);
      else return (-(1));
    }
  }
  return (0);
}
```

Fig. 7: Extracted C code for `wmpn_cmp`.

Lemma 1 (`value_sub_frame`). *Let* $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}$ *such that for all* i, $a_i = b_i$. *Then* $\overline{a_0 \ldots a_{n-1}} = \overline{b_0 \ldots b_{n-1}}$.

The proof is a straightforward induction, which translates well into a Why3 lemma function where the recursive call provides the induction hypothesis.

```
let rec lemma value_sub_frame (x y:map int limb) (n m:int)
  requires { MapEq.map_eq_sub x y n m }
  variant  { m - n }
  ensures  { value_sub x n m = value_sub y n m }
= if n < m then value_sub_frame x y (n+1) m else ()
```

This lemma makes it possible to conclude that the numbers are equal if no difference was found by the end of the loop. Notice that the loop body raises an exception as soon as a difference is found. This emulates the return-inside-a-loop pattern found in imperative languages. At extraction, this pattern is detected and the extracted code simply has a `return` inside the main loop (Section 2.3). Fig. 7 shows the extracted code for the `wmpn_cmp` function.

3.3 Trickier example: long division

Let us now consider an example that showcases one of the many algorithmic tricks from GMP that we ported in our implementation. Long division consists in computing the quotient q and remainder of the division of a big integer a (of size m) by a big integer d of size n. It is a significantly more complex problem than long addition or multiplication. Algorithm 1 is an excerpt of the general case algorithm for long division in GMP (file `mpn/generic/sbpi1_div_qr.c`).

The algorithm consists in computing the limbs of the quotient one by one, starting with the most significant one. The numerator a is overwritten at each

step to contain the partial remainder. At each iteration of the loop (with i decreasing from $m-n$), we compute a quotient limb $\hat{q}$ by dividing the three most significant limbs of the current remainder a (of size $n+i$) by the two most significant limbs from the denominator d. We then subtract from the high part of the current remainder the product of that quotient limb by the denominator. Note that the most significant limb of the current remainder is never stored back to a_{n+i}. It is kept in the local variable x as an optimization.

Algorithm 1 General case long division (abridged).

```
1:  function DIVMOD_GEN(q, a, d, m, n)
2:      ...                                                    ▷ Initialize
3:      while i > 0 do
4:          i ← i − 1
5:          if x = d_{n−1} and a_{n+i−1} = d_{n−2} then
6:              ...                                            ▷ Unlikely special case
7:          else
8:              (q̂, x, l) ← DIV_3BY2(x, a_{n+i−1}, a_{n+i−2}, d_{n−1}, d_{n−2}, v)
9:              b ← SUBMUL_LIMB(a + i, d, n − 2, q̂)
10:             b_1 ← (l < b)          ▷ Last two steps of the subtraction are inlined
11:             a_{i+n−2} ← (l − b mod β)
12:             b_2 ← (x < b_1)
13:             x ← (x − b_1 mod β)                            ▷ Finish subtraction
14:             ...                                            ▷ Adjust
15:         q_i ← q̂
16:     ...                                                    ▷ Finish and return
```

Let us now take a closer look at lines 9 to 13 in Algorithm 1, which expose another optimization of GMP meant to shave a few more processor cycles. The candidate quotient limb $\hat{q}$ is computed at line 8, and we need to subtract the product of this quotient limb and the denominator from the current remainder. This could be done with only the function call at line 9 by passing n instead of $n-2$ (or rather $n-1$ and inlining the last step on x), but we can do better and optimize the last two steps by making use of the remainder that was computed at line 8. Indeed, we can show that the last two steps simply consist in propagating the borrow from the previous subtraction, as the result of the 3 most significant limbs of subtraction is known to be $\overline{lx0}$ in the absence of borrow-in (the post-condition of the division is exactly that $\overline{a_{n+i-2}a_{n+i-1}x} = \hat{q} \times \overline{d_{n-2}d_{n-1}} + \overline{lx}$). Therefore, all that is left to do is propagate the borrow on $\overline{lx0}$. Hence, lines 11 to 15 are equivalent to computing the subtraction

$$\overline{a_i \dots a_{n+i-1}x} - \hat{q} \times \overline{d_0 \dots d_{n-1}}$$

returning b_2 as borrow and writing the result in $\overline{a_i \dots a_{n+i-2}x}$ (one limb fewer). This is exactly the last assertion of Fig. 8, which shows an abridged version of our proof for this part of the algorithm.

```
let a' = C.incr a !i in
let a'' = C.incr a' (Int32.(-) n two) in
label L in
let qu,l,h = div3by2_inv !x (C.get_ofs a'' one) (C.get a'') dh dl v in
let b = submul_limb a' y qu (Int32.(-) n two) in
let b1 = if (Limb.(<) l b) then uone else uzero in
C.set a'' (sub_mod l b);
let b2 = if (Limb.(<) h b1) then uone else uzero in
x := sub_mod h b1;
assert { value a' (n - 1) + power beta (n - 1) * !x
                          - power beta n * b2
         = value (a' at L) n + power beta n * (!x at L)
           - qu * (value d n) };
```

Fig. 8: Transcription (modified for readability) of Algorithm 1, lines 8 to 13.

All in all, this algorithmic trick saves several arithmetic operations: two multiplications, as the two most significant limbs of d are not multiplied by $\hat{q}$, and two subtractions, as in the last two steps, only a carry is propagated instead of doing a subtraction and then propagating a carry. This is far from irrelevant: this loop is the performance-critical one for long integer division, and almost all the cost of the loop is in the `submul_limb` call (it is the only operation with a cost that scales with the size of the input that is run with non-negligible probability). This trick, which makes the cost of the loop similar to what it would be if the denominator was two limbs shorter, illustrates the kind of GMP implementation details that we have to preserve in order to keep up in terms of performance.

3.4 Statistics on the proof effort

We have implemented and verified functions for performing addition, subtraction, multiplication, division, comparison, and logical shifts on arbitrary-precision integers. In many cases, we also provide lower-level functions for the cases when one of the inputs is a single limb or when the two inputs have the same length (equivalent to the functions suffixed by `_1` and `_n` in GMP[6]).

This totals 6000 lines of Why3, which break down into 1350 lines of code and 4650 lines of specifications and proofs, most of which are assertions. The theorem provers Alt-Ergo, CVC3, CVC4, Eprover, and Z3 were used, as well as around ten very short Coq proofs. The total proof time is around 1000 seconds. For a more detailed breakdown, refer to `http://www.lri.fr/~rieu/why3mp/why3session.html`.

[6] `http://gmplib.org/manual/Low_002dlevel-Functions.html`

n \ m	5	7	10	13	15	20
5	0%	7%	8%	6%	8%	12%
7	—	5%	7%	8%	9%	14%
10	—	—	9%	7%	7%	13%
13	—	—	—	9%	7%	14%
15	—	—	—	—	6%	15%
20	—	—	—	—	—	13%

(a) multiplication

n \ m	5	7	10	13	15	20
5	130%	8%	25%	18%	17%	16%
7	—	67%	3%	14%	19%	14%
10	—	—	61%	2%	4%	12%
13	—	—	—	33%	7%	3%
15	—	—	—	—	54%	5%
20	—	—	—	—	—	40%

(b) division

Fig. 9: Overhead for m-by-n operations.

4 Benchmarks

We have compared the execution time of our extracted code against GMP on randomly generated medium-sized integers, up to 1280 bits. For bigger inputs, the comparison becomes increasingly meaningless since GMP switches to divide-and-conquer algorithms which have a better asymptotic complexity. To prevent GMP from using too many architecture-specific optimizations, we have configured GMP with the `--disable-assembly` flag, so that GMP uses only generic C code. This is true both for the arithmetic primitives (which we share with GMP to focus the benchmarks on the algorithms rather than the primitives) but also for the operations on big numbers. Indeed, on many architectures, GMP uses handwritten assembly functions for most of the performance-critical big number algorithms, with performances out of reach of even very efficient C code.

We compare the execution times of GMP (without assembly) and our library on three different functions: addition, multiplication, and division. We do separate measures for all valid combinations of lengths of the input operands between 1 and 20. For each of these, we generate a few thousand random inputs and call each function a hundred times on each input, and record the total time.

For multiplication, our library is between 5 and 10% slower than GMP across all sizes (Fig. 9a). One possible cause for the discrepancy is the use of a different basic block for addition: while we use a primitive that adds two one-limb integers and a carry, GMP uses a primitive that adds two two-limb integers. We intend to switch to GMP's primitive in the near future.

For division, the difference in execution times is much more dependent on the length of the inputs, particularly in the difference in length between numerator and denominator (Fig. 9b). When the length of the denominator is less than half the length of the numerator, our algorithm is quite similar to GMP's and runs in about 20% more time.

The situation changes when the length n of the denominator is more than half the length m of the numerator, that is, more than the length of the quotient. Indeed, GMP no longer applies Algorithm 1 directly on the operands. Instead, the algorithm is called on the $2q$ most significant limbs of the numerator and q

most significant limbs of the denominator, where q is the length of the quotient. This gives an estimated quotient, and a rather involved adjustment step follows. This alternative algorithm is not yet implemented in our library, which simply applies the general algorithm in all cases.

Note that GMP's adjustment step is somewhat expensive in that it requires the allocation of a long integer. Thus, for the small sizes we are considering, the adjustment step seems to dominate the complexity in such a way that the algorithm switch is only worth it when the denominator is almost as long as the numerator. Thus, for $m/2 \leq n < m-1$, the overhead of our library is below 10%. It then increases drastically when the sizes of the numerator and denominator get very close: for $n = m - 1$, our library is around 25% slower than GMP; for $n = m$, our library is sometimes twice as slow. The effect is even stronger for very small inputs.

One more source of overhead for our library is a difference in the function that normalizes the operands and launches the correct division algorithm (`tdiv_qr` in GMP). When dividing a m-limb number $\overline{a}$ by an n-limb one, the quotient sometimes fits on $m - n$ limbs instead of $m - n + 1$ limbs. Additionally, both numerator and denominator are left-shifted so that the denominator becomes normalized (that is, greater than or equal to $\beta/2$). After this shift, the numerator has length either m or $m + 1$, whereas the denominator stays of length n. In the end, passing m as numerator size to the division algorithm is correct only if $a_m = 0$ and the quotient fits on $m - n$ limbs. Otherwise, $m + 1$ has to be passed and a quotient of length $m - n + 1$ is computed. While GMP performs checks to decide whether m or $m + 1$ should be passed, we have not implemented them yet. Our library therefore plays it safe by always passing $m + 1$, which means an extra iteration of the loop in those cases where passing m would have been enough.

5 Related Work

There have been several attempts at formally verifying parts of GMP. Bertot et al. verified the GMP's divide-and-conquer algorithm for square root [3]. It was performed using the Correctness tool which translates a program and its specification into verification conditions for Coq. In that work, the memory is seen as a large array of machine integers, so function specifications have to tell which zones of memory are left unchanged. Other than that, the way the authors implement and specify their square root algorithm is quite close to the way we do ours; thus, had they wished to, they could easily have extracted it to C.

Affeldt verified a binary GCD algorithm and the functions it depends on [1]. Neither multiplications nor divisions are present. The implementation, specification, and verification were done using Coq. An interesting aspect of this work is that the implementation language is a variant of MIPS assembly, so as to effectively produce a low-level verified library. Another interesting point is that, while the GCD algorithm is not GMP's one, the inputs are represented using

GMP's layout for signed integers, which incurs a pointer indirection. To account for this complexity, the author uses a memory model based on separation logic.

Further away from GMP, Berghofer verified an Ada library for performing modular exponentiation [2]. It was written and specified using the SPARK subset of Ada and the verification conditions were then proved using Isabelle/HOL. The use of Montgomery multiplication makes it slightly more complicated than the binary GCD example from an algorithmic point of view. There is no need for a memory model, since arbitrary-precision integers are represented using plain Ada arrays and SPARK prevents them from being aliased.

Fischer designed a modular exponentiation library developed for C and verified using Isabelle/HOL [12]. Multiplication and division algorithms are naive and use arbitrary-precision integers represented using garbage-collected doubly-linked lists of machine integers. Thus, this library is certainly not meant to be efficient. Aliasing issues are solved by using both a Bornat-like memory model [6], so as to automatically distinguish integer words from pointer words, and frame predicates in specifications, so as to declare which heap positions are possibly modified by a function.

Finally, there have been various efforts to verify specific cryptography primitives and their underlying arithmetic. Zinzindohoué et al. verified an elliptic curve library written in F^* and meant to be extracted to C [18]. A peculiarity is that integers are no longer of arbitrary precision; they are represented by fixed-size arrays. Moreover, only part of a machine word is used to store a limb; for instance, a 448-bit integer is stored using 8 limbs of 56 bits (out of 64). As a consequence, arithmetic operations on limbs do not have to be modular (which makes them simpler for SMT solvers to reason about) and carry bits do not have to be propagated. Regarding the memory model, function specifications explicitly tell which parts of the heap are modified.

6 Conclusions

Our work aims at devising a formally verified C library that provides the same arbitrary-precision arithmetic primitives as GMP. At the time of this paper, we have implemented and verified the following algorithms from GMP: comparison, addition, subtraction, multiplication, and division. For multiplication and division, those are only the algorithms meant to be used with integers of size less than 20 limbs, that is, the so-called *schoolbook* algorithms. Moreover, in the case of the division, we are lacking an optimized algorithm when the final quotient is short, which means that the version for computing long quotients is always being called (unless the divisor is one or two limbs long).

Thanks to our memory model and the notion of pointer it provides, we were able to write the functions the same way GMP developers did. It also made it easy to implement an extraction mechanism to C for Why3. Moreover, since this memory model piggybacks on the region mechanism of Why3, we did not have to bother with pointer aliasing, so the specification of the functions is just about their arithmetic properties, contrarily to most of the other verified libraries.

Despite the terminology, the algorithms we have considered are far more intricate than the algorithms one might find in a schoolbook and are still the topic of active research [17]. For instance, the division operator is designed to correctly compute the remainder after a single pass with probability almost 1, and thus does not incur a correction step. Our code implements all the algorithmic tricks that can be found in the corresponding functions of GMP, which makes our library competitive with GMP's non-assembly implementation. In fact, the generated C code is so close to GMP's own code that the formal verification of our library increases the confidence in the correctness of GMP as a by-product.

As it stands, the proof effort for getting a verified GMP-like library is way too costly. Indeed, while the algorithms are highly intricate, the effort required is compounded by the nonlinear nature of the integer properties submitted to the automated solvers. SMT solvers are especially unhelpful there, so the user has to split proofs at a deeper level of detail than what an interactive theorem prover with support for algebraic reasoning would require. Thus, before tackling the implementation and verification of other GMP functions, we intend to work on designing decision procedures dedicated to verifying these arithmetic properties. While the class of nonlinear integer problems is undecidable, the properties that occur when verifying a GMP-like library are sufficiently specific that we have good hope for success.

Once the issue of proof automation has been tackled, we intend to implement and verify divide-and-conquer algorithms for multiplication (e.g. Toom-Cook algorithms) and division, so as to stay competitive with GMP even for larger integers. We also intend to provide the same high-level interface as GMP for abstract signed arbitrary-precision integers. This comes as a new challenge for the memory model, since most `mpz` functions allow for aliasing between their arguments. For instance, one can pass the same large integer as both input and output, so operators have to properly resolve any aliasing issue (e.g. by allocating temporary buffers) before calling into the `mpn` functions.

References

1. Affeldt, R.: On construction of a library of formally verified low-level arithmetic functions. Innovations in Systems and Software Engineering 9(2), 59–77 (2013)
2. Berghofer, S.: Verification of dependable software using SPARK and Isabelle. In: Brauer, J., Roveri, M., Tews, H. (eds.) 6th International Workshop on Systems Software Verification. OpenAccess Series in Informatics (OASIcs), vol. 24, pp. 15–31. Dagstuhl, Germany (2012)
3. Bertot, Y., Magaud, N., Zimmermann, P.: A proof of GMP square root. Journal of Automated Reasoning 29(3-4), 225–252 (2002)
4. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64. Wrocław, Poland (August 2011), `http://proval.lri.fr/publications/boogie11final.pdf`
5. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with Why3. International Journal on Software Tools for Technology Transfer (STTT) 17(6), 709–727 (2015), see also `http://toccata.lri.fr/gallery/fm2012comp.en.html`

6. Bornat, R.: Proving pointer programs in Hoare logic. In: Mathematics of Program Construction. pp. 102–126 (2000)
7. Filliâtre, J.C.: One logic to use them all. In: 24th International Conference on Automated Deduction (CADE-24). Lecture Notes in Artificial Intelligence, vol. 7898, pp. 1–20. Springer, Lake Placid, USA (June 2013)
8. Filliâtre, J.C., Gondelman, L., Paskevich, A.: A pragmatic type system for deductive verification. Research report, Université Paris Sud (2016), `https://hal.archives-ouvertes.fr/hal-01256434v3`
9. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The spirit of ghost code. Formal Methods in System Design 48(3), 152–174 (2016)
10. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) 19th International Conference on Computer Aided Verification. Lecture Notes in Computer Science, vol. 4590, pp. 173–177. Springer, Berlin, Germany (Jul 2007), `http://www.lri.fr/~filliatr/ftp/publis/cav07.pdf`
11. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (Mar 2013)
12. Fischer, S.: Formal verification of a big integer library. In: DATE Workshop on Dependable Software Systems (2008), `http://www-wjp.cs.uni-sb.de/publikationen/Fi08DATE.pdf`
13. Fumex, C., Dross, C., Gerlach, J., Marché, C.: Specification and proof of high-level functional properties of bit-level programs. In: 8th NASA Formal Methods Symposium. Lecture Notes in Computer Science, vol. 9690. Springer, Minneapolis, MN, USA (Jun 2016)
14. International Organization for Standardization: ISO/IEC 9899:1999: Programming Languages – C (2000)
15. Kosmatov, N., Marché, C., Moy, Y., Signoles, J.: Static versus dynamic verification in Why3, Frama-C and SPARK 2014. In: 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. pp. 461–478. Lecture Notes in Computer Science, Springer (Oct 2016)
16. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Usable Verification Workshop (2010), `http://fm.csl.sri.com/UV10/`
17. Moller, N., Granlund, T.: Improved division by invariant integers. IEEE Transactions on Computers 60(2), 165–175 (2011)
18. Zinzindohoué, J.K., Bartzia, E.I., Bhargavan, K.: A verified extensible library of elliptic curves. In: Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF). pp. 296–309 (2016)