# How to get an efficient yet verified arbitrary-precision integer library


Raphaël Rieu-Helft

(joint work with Guillaume Melquiond and Claude Marché)

TrustInSoft Inria


November 13, 2018

# Context, motivation, goals

goal: efficient and formally verified large-integer library

GMP:

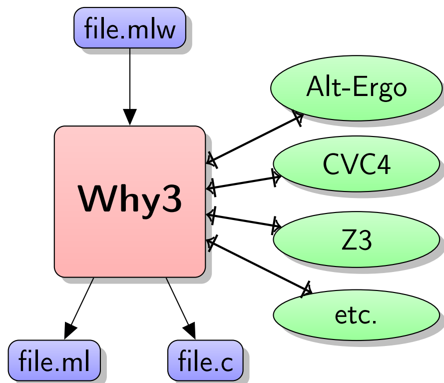
- widely-used, high-performance library
- tested, but hard to ensure good coverage (unlikely branches)
- correctness bugs have been found in the past

idea:

1. formally verify GMP algorithms with Why3
2. extract efficient C code

# Reimplementing GMP using Why3

# General approach




game plan:

- implement the GMP algorithms in WhyML
- verify them with Why3
- extract to C

difficulties:

- preserve all GMP implementation tricks
- prove them correct
- extract to efficient C code

# An example: comparison

large integer $\equiv$ pointer to array of unsigned integers $a_0 \dots a_{n-1}$ called limbs

$$\texttt{value}(a, n) = \sum_{i=0}^{n-1} a_i \beta^i \qquad \text{usually } \beta = 2^{64}$$

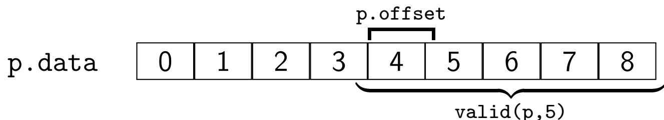
```
type ptr 'a = ...
let wmpn_cmp (x y:ptr limb) (sz:int32) : int32
=
 let ref i = sz in
 while i ≥ 1 do
   i ← i - 1;
   let lx = x[i] in
   let ly = y[i] in
   if lx ≠ ly
   then
      if lx > ly
      then return 1
      else return -1
   end
 done;
 0
```

# Memory model

simple memory model, more restrictive than C

```
type ptr 'a = abstract { mutable data: array 'a ; offset: int }

predicate valid (p:ptr 'a) (sz:int) =
  0 ≤ sz ∧ 0 ≤ p.offset ∧ p.offset + sz ≤ plength p
```




```
val malloc (sz:uint32) : ptr 'a                    (*   malloc(sz * sizeof('a))   *)
  ...

val free (p:ptr 'a) : unit                         (*   free(p)   *)
  ...
```

no explicit address for pointers

# Alias control

aliased C pointers ⇔ point to the same memory object
aliased Why3 pointers ⇔ same `data` field

only way to get aliased pointers: `incr`

```
type ptr 'a = abstract { mutable data: array 'a ; offset: int }

val incr (p:ptr 'a) (ofs:int32): ptr 'a          (*   p+ofs   *)
  alias    { result.data with p.data }
  ensures  { result.offset = p.offset + ofs }
  ...

val free (p:ptr 'a) : unit
  requires { p.offset = 0 }
  writes   { p.data }
  ensures  { p.data.length = 0 }
```

Why3 type system: all aliases are known statically
⇒ no need to prove non-aliasing hypotheses

# Example specification: long multiplication

specifications are defined in terms of `value`

```
(** [wmpn_mul r x y sx sy] multiplies [(x, sx)] and [(y,sy)] and writes
the result in [(r, sx+sy)]. [sx] must be greater than or equal to [sy].
Corresponds to [mpn_mul]. *)
let wmpn_mul (r x y: ptr uint64) (sx sy: int32) : unit
  requires { 0 < sy ≤ sx }
  requires { valid x sx }
  requires { valid y sy }
  requires { valid r (sy + sx) }
  writes { r.data.elts }
  ensures { value r (sy + sx) = value x sx * value y sy }
```

Why3 typing constraint: `r` cannot be aliased to `x` or `y`

- simplifies proof: aliases are known statically
- we need separate functions for in-place operations

## An example: schoolbook multiplication

# Schoolbook multiplication

- simple algorithm, optimal for smaller sizes
- GMP switches to divide-and-conquer algorithms at $\sim 20$ words

```
mp_limb_t
mpn_mul (mp_ptr rp, mp_srcptr up, mp_size_t un, mp_srcptr vp,
         mp_size_t vn)
{
  /* We first multiply by the low order limb. This result can be
     stored, not added, to rp. We also avoid a loop for zeroing this
     way. */

  rp[un] = mpn_mul_1 (rp, up, un, vp[0]);

  /* Now accumulate the product of up[] and the next higher limb from
     vp[]. */

  while (--vn >= 1)
    {
      rp += 1, vp += 1;
      rp[un] = mpn_addmul_1 (rp, up, un, vp[0]);
    }
  return rp[un];
}
```

# Why3 implementation

```
while i < sy do
  invariant { value r (i + sx) = value x sx * value y i }




  ly ← get_ofs y i;
  let c = addmul_limb rp x ly sx in


  set_ofs rp sx c;
  i ← i + 1;



  rp ← C.incr rp 1;
done;
...
```

# Why3 implementation

```
while i < sy do
  invariant { 0 ≤ i ≤ sy }
  invariant { value r (i + sx) = value x sx * value y i }
  invariant { (rp).offset = r.offset + i }
  invariant { plength rp = plength r }
  invariant { pelts rp = pelts r }
  variant { sy - i }
  ly ← get_ofs y i;
  let c = addmul_limb rp x ly sx in
  value_sub_update_no_change (pelts r) ((rp).offset + sx)
                             r.offset (r.offset + i) c;
  set_ofs rp sx c;
  i ← i + 1;
  value_sub_tail (pelts r) r.offset (r.offset + sx + k);
  value_sub_tail (pelts y) y.offset (y.offset + k);
  value_sub_concat (pelts r) r.offset (r.offset + k) (r.offset + k + sx);
  rp ← C.incr rp 1;
done;
...
```

# Building block: addmul_limb

```
(** [addmul_limb r x y sz] multiplies [(x, sz)] by [y], adds the [sz]
    least significant limbs to [(r, sz)] and writes the result in [(r,sz)].
    Returns the most significant limb of the product plus the carry
    of the addition. Corresponds to [mpn_addmul_1].*)

let addmul_limb (r x: ptr uint64) (y: uint64) (sz: int32): uint64
  requires { valid x sz }
  requires { valid r sz }
  ensures  { value r sz + (power radix sz) * result
             = value (old r) sz + value x sz * y }
  writes   { r.data.elts }
  ensures  { forall j. j < r.offset ∨ r.offset + sz ≤ j → r[j] = (old r)[j] }
```

- adds $y \times \overline{x}$ to $\overline{r}$
- does not change the contents of r outside the first sz cells
- called on $\mathtt{r} + i$, x and $\mathtt{y}_i$ for $0 \leq i \leq \mathtt{sy}$

# Extracted code

```
void wmpn_mul_basecase(uint64_t * r, uint64_t * x, uint64_t * y,
                       int32_t sx, int32_t sy)
{
  uint64_t ly;
  uint64_t c;
  uint64_t * rp;
  int32_t i;
  uint64_t res;
  ly = (*y);
  c = wmpn_mul_1(r, x, ly, sx);
  r[sx] = c;
  rp = (r + 1);
  i = 1;
  while ((i) < sy) {
    ly = (y[(i)]);
    res = wmpn_addmul_1(rp, x, ly, sx);
    (rp)[sx] = res;
    i = ((i) + 1);
    rp = (rp) + 1;
  }
}
```

not as concise as GMP, but close enough to be optimized by the compiler

# Algorithms, benchmarks

# Schoolbook algorithms

- comparison
- addition/subtraction
  $\Rightarrow$ many variants (in-place, with/without carry checking...)
- multiplication
  $\Rightarrow O(n^2)$: used for operands of less than 30 limbs
- logical shifts

Total effort: $\sim$ 1000 lines of programs, $\sim$ 1100 lines of specs/proofs

# Division

Heavily optimised schoolbook algorithm

- Use of 3-by-2 division to compute each quotient limb
  $\Rightarrow$ fewer adjustment steps
- Fast 3-by-2 divisions using a pseudo-inverse and no division primitives
  (Möller & Granlund 2011)

Total effort: $\sim$ 750 lines of programs, $\sim$ 3300 lines of specs/proofs

# Toom-Cook multiplication

Divide-and-conquer multiplication algorithm

$O(n^k)$, $1 < k < 2$

Suitable for operands of 30-100 limbs

Two mutually recursive variants:

- Toom-2: split each operand in 2 parts ($\sim$ Karatsuba)
- Toom-2.5: split large operand in 3 parts and small in 2

Total effort: $\sim$ 900 lines of programs, $\sim$ 1300 lines of specs/proofs

# Comparison with GMP

we compare with GMP without assembly (option `--disable-assembly`)

multiplication: less than 5% slower than GMP

division: $\sim 10\%$ slower than GMP

- except for very small inputs
- except for `sx` very close to `sy`
  $\Rightarrow$ GMP uses a different algorithm, not ported yet

# Proof effort

- 9000 lines of Why3 code
  - 3000 of programs
  - 6000 of specifications and (mostly) assertions

large proof contexts, nonlinear arithmetic
$\Rightarrow$ many long assertions are needed even for some "easy" goals

Ongoing: use computational reflection to automate some future proofs and delete some existing assertions
$\Rightarrow \sim 700$ lines of assertions deleted

# Conclusions

verified C library, bit-compatible with GMP's `mpn` layer

GMP implementation tricks preserved
$\Rightarrow$ satisfactory performances in the handled cases

new Why3 features:

- extraction and memory model for C
- `alias` of return value and parameter
- Why3 framework for proofs by reflection

coming soon:

- divide-and-conquer division, square root, modular exponentiation
- cryptographic primitives (side-channel resistant)
- GMP `mpz` layer