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

Raphaël Rieu-Helft

(joint work with Guillaume Melquiond and Claude Marché)

TrustInSoft Inria

April 20, 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

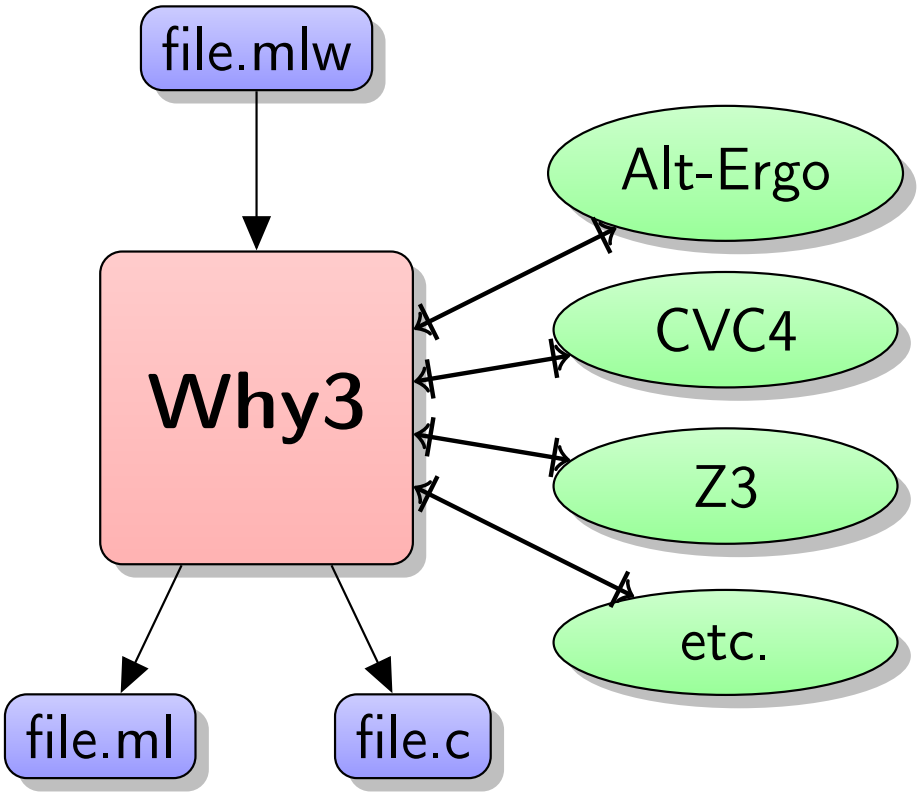
idea:

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

Outline

Reimplementing GMP using Why3

Tool: the Why3 platform



approach:

- 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 $a_0 \ldots a_{n-1}$ called limbs

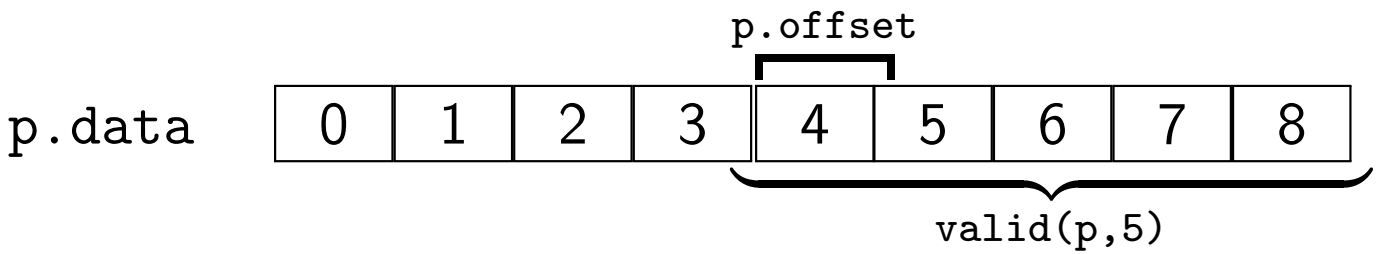
$$\mathrm{value}(a, n) = \sum_{i=0}^{n-1} a_i \beta^i \qquad \text{usually } \beta = 2^{64}$$

```
type ptr 'a = ...
exception Return32 int32

let wmpn_cmp (x y: ptr uint64) (sz: int32): int32
= let i = ref sz in
  try
    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 raise (Return32 1)
        else raise (Return32 (-1))
    done;
    0
  with Return32 r → r
  end
```

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 $\Leftrightarrow$ point to the same memory object
aliased Why3 pointers $\Leftrightarrow$ 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
$\Rightarrow$ 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 proofs : aliases are known statically
- we need separate functions for in-place operations

Extraction mechanism

goals:

- simple, straightforward extraction (trusted)
- performance: no added complexity, no closures or indirections
- inefficiencies caused by extraction must be optimizable by the compiler

tradeoff: handle only a small, C-like fragment of WhyML

- ✓ loops
- ✓ references
- ✓ machine integers
- ✓ manual memory management
- ✗ polymorphism, abstract types
- ✗ higher order
- ✗ mathematical integers
- ✗ garbage collection

Comparison: extracted C code

```
let wmpn_cmp (x y: ptr uint64)
             (sz: int32): int32
= let i = ref sz in
  try
    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 raise (Return32 1)
        else raise (Return32 (-1))
    done;
    0
  with Return32 r → r
  end
```

```
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);
}
```

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);
  assert { value r (!i + sx) = value x sx * value y !i
           by ...
              so ...
              so ... (* 20+ subgoals *) };
  !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 mul(uint64_t * r12, uint64_t * x15, uint64_t * y13,
         int32_t sx4, int32_t sy4)
{
  uint64_t ly9, c8, res16;
  uint64_t * rp3;
  int32_t i16, o28;
  uint64_t * o29;
  ly9 = (*(y13));
  c8 = (mul_limb(r12, x15, ly9, sx4));
  *(r12+(sx4)) = c8;
  rp3 = (r12+(1));
  i16 = (1);
  while (i16 < sy4) {
      ly9 = *(y13+(i16));
      res16 = (addmul_limb(rp3, x15, ly9, sx4));
      *(rp3+(sx4)) = res16;
      o28 = (i16 + 1); i16 = o28;
      o29 = (rp3+(1)); (rp3) = o29;
    }
  return (*(rp3+(sx4 - 1)));
}
```

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

Benchmarks, conclusions

Comparison with GMP

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

we only consider inputs of 20 words or less ($\sim$ 1300 bits)
$\Rightarrow$ above that, GMP uses different algorithms

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 for `sy` $>$ `sx`$/2$, to do

performances are very dependent on the compiled code of the primitives
ongoing: link to GMP to use the exact same primitives

Proof effort

- 6000 lines of Why3 code
  - 1350 of programs
  - 4650 of specifications and (mostly) assertions
- 4200 subgoals, around two thirds are for division

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

Ongoing: use computational reflection to automate some proofs and delete the assertions
$\Rightarrow \sim 700$ lines of assertions deleted, work in progress
$\Rightarrow$ removes the need for some tedious proofs, but still finnicky

Conclusions

verified C library, bit-compatible with GMP
GMP `mpn` functions implemented: schoolbook add, sub, mul, div, shifts, divide-and-conquer multiplication (wip)

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 algorithms for multiplication and division
- GMP `mpz` functions
- extract specifications as well?

Arithmetic primitives: the uint64 type

```
type uint64
val function to_int (n:uint64): int
meta coercion function to_int

let constant max = 0xffff_ffff_ffff_ffff
predicate in_bounds (n:int) = 0 ≤ n ≤ max
axiom to_int_in_bounds: forall n:uint64. in_bounds n

val mul (x y:uint64): uint64                          (* defensive, no overflow *)
  requires { in_bounds (x * y)  }
  ensures  { result = x * y }

val mul_mod (x y:uint64): uint64                      (* with overflow *)
  ensures { result = mod (x * y) (max+1) }

val mul_double (x y:uint64): (uint64,uint64)          (* returns two words*)
  returns { (l,h) → l + (max+1) * h = x * y }
```