A Why3 framework for reflection proofs and its application to GMP's algorithms

Guillaume Melquiond[1] Raphaël Rieu-Helft[2,1]

[1]Inria [2]TrustInSoft

July 15, 2018

Context, motivation, goals

goal: efficient and formally verified large-integer library

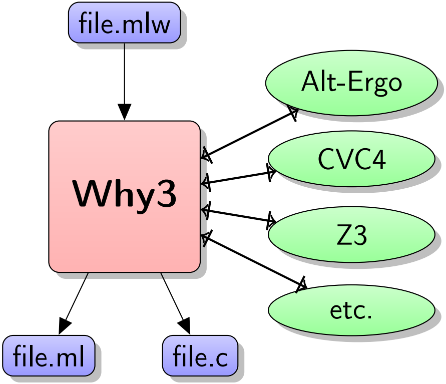
GMP:

- widely-used, high-performance library
- safety-critical
- 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

Tool: the Why3 platform



approach:

- implement GMP algorithms and their specifications in WhyML
- prove Why3-generated verification conditions
- extract to C

main challenge: how to keep proof effort manageable?

Reimplementing GMP using Why3

An example: comparison

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

$$\overline{a_0 \dots a_{n-1}} = \sum_{i=0}^{n-1} a_i \beta^i \qquad \text{usually } \beta = 2^{64}$$

```
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
```

(R. Rieu-Helft, C. Marché, G. Melquiond, *How to Get an Efficient yet Verified Arbitrary-Precision Integer Library*, VSTTE'17)

Example specification: long addition

specifications are defined in terms of the function $\texttt{value}(a, n) = \sum_{i=0}^{n-1} a_i \beta^i$

```
(** `wmpn_add_n r x y sz` adds `(x, sz)` to `(y, sz)` and writes the
    result in `(r, sz)`. Returns carry, either 0 or 1.
    Corresponds to `mpn_add_n`. *)
let wmpn_add_n (r x y: ptr limb) (sz: int32) : limb
  requires { valid x sz }
  requires { valid y sz }
  requires { valid r sz }
  ensures  { value r sz + (power radix sz) * result
             = value x sz + value y sz }
  ensures  { 0 ≤ result ≤ 1 }
  writes   { r.data.elts }
```

Why3 implementation

```
while !i < sz do

  invariant { 0 ≤ !i ≤ sz }
  invariant { value r !i + (power radix !i) * !c =
              value x !i + value y !i }
  invariant { 0 ≤ !c ≤ 1 }

  lx := x[!i];
  ly := y[!i];
  let res, c1 = add_with_carry !lx !ly !c in
  r[!i] ← res;

  c := c1;









  i := !i + 1;
done;
!c
```

Why3 implementation

```
while !i < sz do
  variant { sz - !i }
  invariant { 0 ≤ !i ≤ sz }
  invariant { value r !i + (power radix !i) * !c =
              value x !i + value y !i }
  invariant { 0 ≤ !c ≤ 1 }
  label StartLoop in
  lx := x[!i];
  ly := y[!i];
  let res, c1 = add_with_carry !lx !ly !c in
  r[!i] ← res;
  assert { value r !i = (value r !i at StartLoop) };
  c := c1;
  value_tail r !i;
  value_tail x !i;
  value_tail y !i;
  assert { value r (!i+1) + (power radix (!i+1)) * !c =
           value x (!i+1) + value y (!i+1)
           by ...
           so ...(* 10+ lines *)  };
  i := !i + 1;
done;
!c
```

Motivation

total proof effort (add, sub, mul, div, logical shifts) (VSTTE'17) :

$\sim$ 6000 lines of Why3 code

- $\sim$ 1500 of programs
- $\sim$ 500 of specifications
- $\sim$ 4000 of proof cuts

This is too much work!

SMT solvers fail because of large proof contexts, nonlinear arithmetic...
$\Rightarrow$ many long assertions are needed even for some “easy” goals

Zooming in

```
assert { value r (!i+1) + (power radix (!i+1)) * !c =
         value x (!i+1) + value y (!i+1) };
```

Generated verification condition:

```
H:  value r1 i + (power radix i) * c1 = value x i + value y i   ×1
H1: res + radix * c = lx + ly + c1                   × power radix i
H2: value r i = value r1 i                                      ×1
H3: value x (i+1) = value x i + (power radix i) * lx        ×(−1)
H4: value y (i+1) = value y i + (power radix i) * ly        ×(−1)
H5: value r (i+1) = value r i + (power radix i) * res           ×1
-------------------------------------------------------------------------------
g:  value r (i+1) + power radix (i+1) * c = value x (i+1) + value y (i+1)
```

the goal is actually a linear combination of the hypotheses

1. Reimplementing GMP using Why3
2. Computational reflection in Why3
3. Effectful programs as decision procedures

Computational reflection in Why3

Toy example: equality in a ring

goal: prove equalities such as $\mathbf{M} = \mathbf{M}'$ with

$$\begin{aligned}
\mathbf{M} &= \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} \\
\mathbf{M}' &= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2}) \cdot (\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) + \mathbf{A}_{2,2} \cdot (\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\
&\quad - (\mathbf{A}_{1,1} + \mathbf{A}_{1,2}) \cdot \mathbf{B}_{2,2} + (\mathbf{A}_{1,2} - \mathbf{A}_{2,2}) \cdot (\mathbf{B}_{2,1} + \mathbf{B}_{2,2})
\end{aligned}$$

SMT solvers time out in practice

idea: embed terms into the logical language of Why3

```
type t = Var int | Add t t | Mul t t | Sub t t

let rec function interp (x: t) (y: int → a) : a =
  match x with
  | Var n → y n
  | Add x1 x2 → (interp x1 y) + (interp x2 y)
  | Mul x1 x2 → (interp x1 y) * (interp x2 y)
  | Sub x1 x2 → (interp x1 y) - (interp x2 y)
  end
```

Decision procedures

```
function eq_zero (x:t) : bool
= match x with
  ... (* purely functional code, structurally decreasing arguments *)

lemma zero_sub_eq:
  forall x1 x2 y. eq_zero (Sub x1 x2) → interp x1 y = interp x2 y
```

`eq_zero` computes a normal form, but no need to prove (or define) that
⇒ this proof is very easy

to instantiate the lemma, we need to guess `x1`, `x2`, `y` such that

$$\texttt{interp x1 y} = \mathsf{M} \qquad \texttt{interp x2 y} = \mathsf{M}'$$

Reification

heuristic approach: invert `zero_sub_eq` and the body of `interp`

```
type t = Var int | Add t t | Mul t t | Sub t t

let rec function interp (x: t) (y: int → a) : a =
  match x with
  | Var n → y n
  | Add x1 x2 → (interp x1 y) + (interp x2 y)
  | Mul x1 x2 → (interp x1 y) * (interp x2 y)
  | Sub x1 x2 → (interp x1 y) - (interp x2 y)
  end
lemma zero_sub_eq:
  forall x1 x2 y. eq_zero (Sub x1 x2) → interp x1 y = interp x2 y

goal g: foo a + b = c * b
```

```
[foo a + b = c * b]
[foo a + b] = [c * b]
Add [foo a] [b] = [c * b]
Add (Var 0) (Var 1) = [c * b]
Add (Var 0) (Var 1) = Mul [c] [b]
Add (Var 0) (Var 1) = Mul (Var 2) (Var 1)
```

```
y 0 = foo a
y 1 = b
y 2 = c
```

Extension: reifying the proof context

```
function interp_eq (g:equality) (y:vars) (z:C.cvars) : bool
= match g with (g1, g2) → interp g1 y z = interp g2 y z end

function interp_ctx (l:list equality) (g:equality) (y:vars) (z:C.cvars) : bool
= match l with
  | Nil → interp_eq g y z (* goal *)
  | Cons h t → (interp_eq h y z) → (interp_ctx t g y z)
  end
```

- recognize implication and recursive call in the `Cons` branch
- one element of the list = one hypothesis in the proof context
- heuristic: match all possible hypotheses in the proof context against the left-hand side

Reflection as a Why3 transformation

```
lemma zero_sub_eq:
  forall x1 x2 y. eq_zero (Sub x1 x2) → interp x1 y = interp x2 y
```

synopsis:

- guess appropriate values for parameters using the reification procedure
- ask the user to prove the premises
- add the instantiated conclusion to the proof context

if we guess wrong, proof probably fails, but no soundness issue
⇒ no need to trust the reification procedure

Effectful programs as decision procedures

From logic to programs

important limitation of computations within the Why3 logic:

- no arrays, loops, references, exceptions...
- must prove termination with structurally decreasing argument

consequence: decision procedures are hard to implement and inefficient

idea: write decision procedures as regular, proved Why3 programs

Interpreter

additional step of the reflection transformation: compute the results

new interpreter for WhyML programs

- based on the intermediate language of Why3's extraction
- simple, but part of the trusted computing base

Example: systems of linear equalities

```
type expr = Term coeff int | Add expr expr | Cst coeff
type equality = (expr, expr)

let linear_decision (l: list equality) (g: equality) : bool
  requires { valid_ctx l ∧ valid_eq g }
  ensures { forall y z. result = True → interp_ctx l g y z }
  raises  { Unknown → true }
= let m = Matrix.make ...
  ... (* exceptions, loops, side effects, mutable states... *)
  match gauss_jordan m with
    | Some r → check_combination l g r
    | None → False
  end
```

- given a list `l` of valid equalities, is the equality g valid?
- check if g is a linear combination of `l` by Gaussian elimination
- proof by certificate: no need to prove Gaussian elimination correctness
- generic: only requires `coeff` to provide partial field operations

Specialized coefficients for GMP goals

```
H:  value r1 i + (power radix i) * c1 = value x i + value y i   ×1
H1: res + radix * c = lx + ly + c1                           × power radix i
...
```

the (symbolic) powers of `radix` need to be part of the scalar coefficients

```
type exp = Lit int | Var int | Plus exp exp | Minus exp | Sub exp exp
type rat = (int, int)
type coeff = (rat, exp)

function qinterp (q:rat) : real
= let (n,d) = q in from_int n /. from_int d

function interp_exp (e:exp) (y:vars) : int
= match e with
  | Lit n → n | Var v → y v
  | Plus e1 e2 → interp_exp e1 y + interp_exp e2 y
  ...
  end

function interp (t:coeff) (y:vars) : real
= let (q,e) = t in qinterp q *. pow radix (from_int (interp_exp e y))
```

Assessment

- user-supplied Why3 programs can be used as decision procedures
- no need to know the internal workings of Why3
- compositionality: existing procedures can be adapted and reused
- minimal impact on the trusted computing base

GMP proof: $\sim$ 1000 lines of assertions can be deleted

main limitation: still hard to debug when it doesn't work

Conclusion

main contributions:

- reflection framework (reification + interpreter)
- a Why3 decision procedure for systems of linear equalities
- much more automatic proofs for GMP algorithms

future work:

- more decision procedures for GMP (inequalities, divisibility. . .)
- improve user experience (what to do when proof fails?)