Functional correctness of an optimized modular inversion algorithm

Tomás Vallejos Parada[1]
joint work with
Assia Mahboubi[1,2] Guillaume Melquiond[1] Pierre-Yves Strub[3]

Inria[1], France

Vrije Universiteit Amsterdam[2], The Netherlands

PQShield[3], France

April 9, 2026

Outline

Definition

The modular inverse of y modulo m is a number y^{-1}, such that $y \cdot y^{-1} = 1 (\text{mod } m)$ holds.

- Used on cryptographic algorithm families such as RSA and ECC.
- Expensive operation
- Two approaches:= Fermat's little theorem ($a^{-1} \equiv a^{p-2} (mod p)$) and extended binary (euclidean) greatest common divisor (GCD).

Today I focus on a modular inverse based on extended binary GCD.

How does GCD compute modular inversion?

A classical modular inverse

$ModInv(y\ m : int) :=$
$a \leftarrow y,\ \ u \leftarrow 1,\ \ b \leftarrow m,\ \ v \leftarrow 0$
while $a \neq 0$
 if $a = 0 \bmod 2$
  $a \leftarrow a/2$
  $u \leftarrow u/2 \bmod m$
 else
  if $a < b$
   $(a, u, b, v) \leftarrow (b, v, a, u)$
  $a \leftarrow (a - b)/2$
  $u \leftarrow (u - v)/2 \bmod m$
return v

Lemma

For all natural numbers y, m, such that,
$1 \leq y < m, prime\ m, m \geq 3$, then
$ModInv(y, m) \cdot y = 1 (mod\ m)$

Proof

It suffices to show that after the execution of the loop

1. a is 0.
2. $GCD(y, m) = GCD(a, b)$.
3. $b = v \cdot y (mod\ m)$.

Pornin's modular inversion algorithm

Pornin's modular inversion algorithm is based on extended binary GCD.
It introduces three optimization ingredients:

- Constant-time algorithm
- Mutualizing updates of u and v
- Approximating computation on a and b

Goal

Proof an optimization for *ModInv* correct, reusing the proof we know.

Strategy

1. Algorithm A with proof P requiring hypotheses $H_1, ..., H_n$
2. Modify A in such a way that most of the proof doesn't break
3. Enjoy of having a proof for modified A by proving the broken hypotheses
4. Move on to the next optimization!

Scheme for ModInv

A classical modular inverse

$ModInv(y\ m : int) :=$
$a \leftarrow y,\ u \leftarrow 1,\ b \leftarrow m,\ v \leftarrow 0$
while $a \neq 0$
 if $a = 0 \bmod 2$
  $a \leftarrow a/2$
  $u \leftarrow u/2 \bmod m$
 else
  if $a < b$
   $(a, u, b, v) \leftarrow (b, v, a, u)$
  $a \leftarrow (a - b)/2$
  $u \leftarrow (u - v)/2 \bmod m$
return v

To prove optimizations

1. Constant-time algorithm
2. Mutualizing updates of u and v
3. Approximating computation on a and b

Still suffices ...

1. a is 0.
2. $GCD(y, m) = GCD(a, b)$.
3. $b = v \cdot y (mod\ m)$.

Unraveling the invariants for each optimization

Constant-time

Coefficient's mutualization

Computation approximation

Reminder

It suffices to show that after the execution of the loop

1. **a is 0.**
2. $GCD(y, m) = GCD(a, b)$.
3. $b = v \cdot y (mod\ m)$.

Warning

The fully optimized algorithm is about to appear.

```
let function PorninGcd (y m k : int) =
  (a, u, b, v) ←(y, 1, m, 0) (ū, v̄) ←(u, v)
  for i = 1 to ⌈(2len(m) − 1)/(k − 1)⌉
    n ←max(len(a), len(b), 2k)
    ā ← 2^(k−1)⌊a/2^(n−k−1)⌋ + (a mod 2^(k−1))
    b̄ ← 2^(k−1)⌊b/2^(n−k−1)⌋ + (b mod 2^(k−1))
    (f0, g0, f1, g1) ←(1, 0, 0, 1) (f̄0, ḡ0, f̄1, ḡ1) ←(f0, g0, f1, g1)
    for j = 1 to k − 1
      if mod ā 2 = 0 then
        ā ← ā/2 (ū, f̄0, ḡ0) ←(div2m ū, div2m f̄0, div2m ḡ0)
      else
        if ā < b̄ then
          (ā, b̄) ← (b̄, ā)  (ū, v̄) ← (v̄, ū)
          (f0, g0, f1, g1) ←(f1, g1, f0, g0) (f̄0,ḡ0,f̄1,ḡ1)←(f̄1,ḡ1,f̄0,ḡ0)
        ā ← (ā - b̄)/2  ū ← div2m (ū - v̄)
        (f0, g0) ←(f0 - f1, g0- g1) (f̄0, ḡ0)←(div2m (f̄0-f̄1), div2m (ḡ0-ḡ1))
      (f1, g1) ←(2f1, 2g1)
    (a, b) ←((a f0+ b g0)/2^(k−1), (a f1+ b g1)/2^(k−1))
    if a < 0 then
      (a, f0, g0) ←(-a, -f0, -g0)
    if b < 0 then
      (b, f1, g1) ←(-b, -f1, -g1)
    (u, v) ←((u f0+ v g0) mod m, (u f1+ v g1) mod m)
  v ←v/(2^((k−1)(⌈2len(m)−1)/(k−1)⌉)) mod m
  if b ≠ 1 then return 0 else return v
```

Zooming into the approximation

$$\bar{a} \leftarrow 2^{k-1}\lfloor a/2^{n-k-1}\rfloor + (a \bmod 2^{k-1})$$

$$a = \underbrace{a_0 \quad \cdots \quad a_{k+1}}_{\substack{k+1 \\ 2^{k-1}\lfloor a/2^{n-k-1}\rfloor}} \quad \cdots \quad \underbrace{a_{n-k+1} \quad \cdots \quad a_n}_{\substack{k-1 \\ a \bmod 2^{k-1}}}$$

$$\bar{a} = \underbrace{a_0 \quad \cdots \quad a_{k+1} \quad a_{n-k+1} \quad \cdots \quad a_n}_{2k}$$

The issue: proving that a decreases enough during the inner iterations

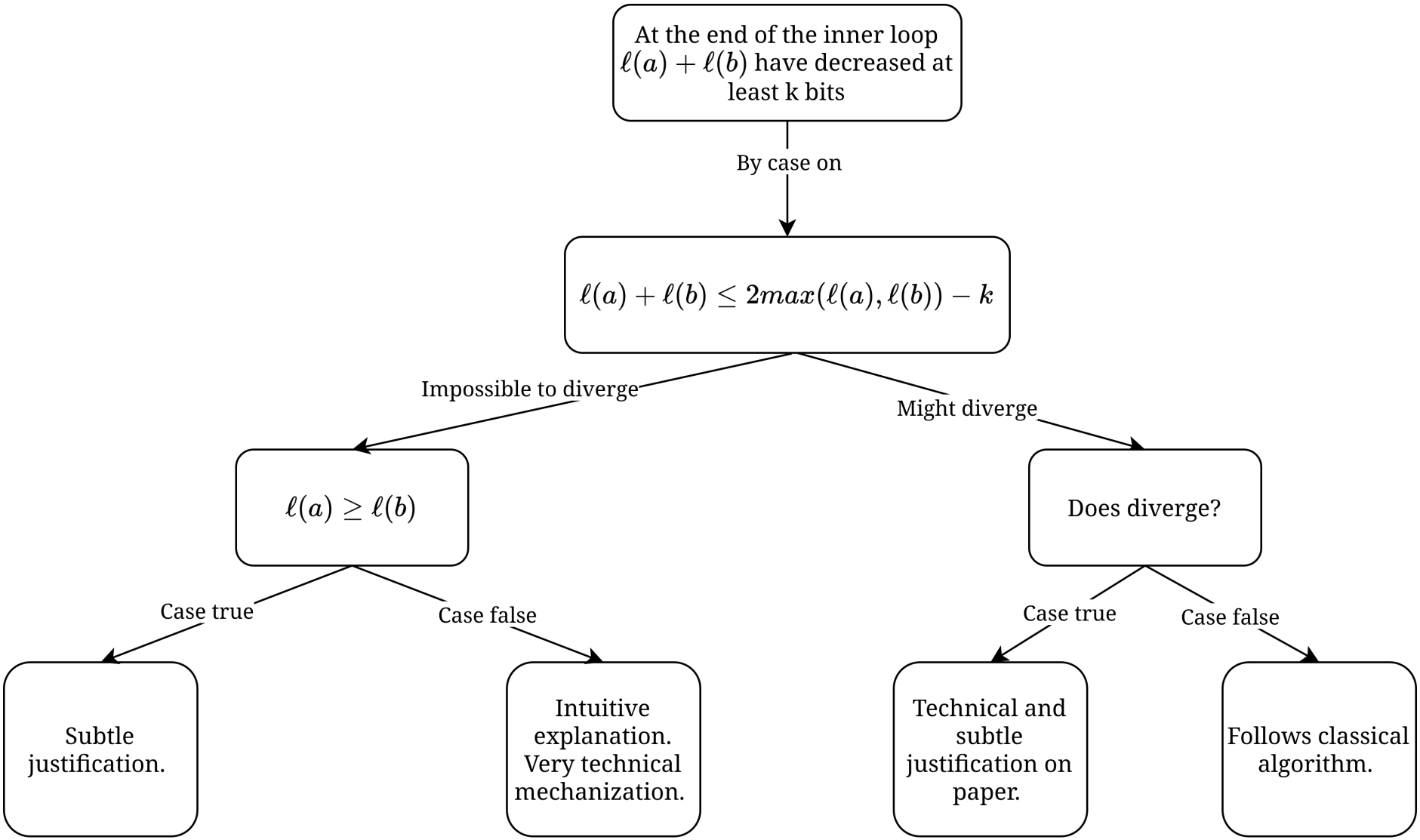
An iteration is divergent if $(\bar{a} < \bar{b} \wedge a \geq b) \vee (\bar{a} \geq \bar{b} \wedge a < b)$
Starting point: a pen and paper proof analyzing the behavior of the algorithm by cases.

First difficulty: incomplete original proof

Divergence-free Hidden technical justification for inconvenient case.
Divergence Assumption of non-trivial hypotheses which are not discharged.
Post-divergence Non-trivial cases are missing in the analysis.

We complete the proof and develop a mechanization blueprint.

Second difficulty: proof structure



Divergence proof sketch

Under hypotheses:

- $\ell(a) + \ell(b) \geq 2max(\ell(a), \ell(b)) - k + 1$
- inner iteration diverges at step d

$\ell(a) + \ell(b)$ decreases by at least k bits by the end of the inner loop.
By proving that:

1. At step d, at least d bits were lost.
2. At step $d + 1$ enough bits are lost.
3. $H(t) \triangleq P(t) \vee Q(t)$ is an invariant post-divergence.

With

$$P(t) \triangleq M_t \geq 2^{n-k+1} \wedge -2^{n-k} < m_t \leq 0$$

$$Q(t) \triangleq \max(|a_t|, |b_t|) < 2^{n-k+1}$$

$$M_t \triangleq \max(a_t, b_t) \quad \text{and} \quad m_d \triangleq \min(a_t, b_t)$$

Mechanization overview

Collaboration between Easycrypt, Rocq and Why3

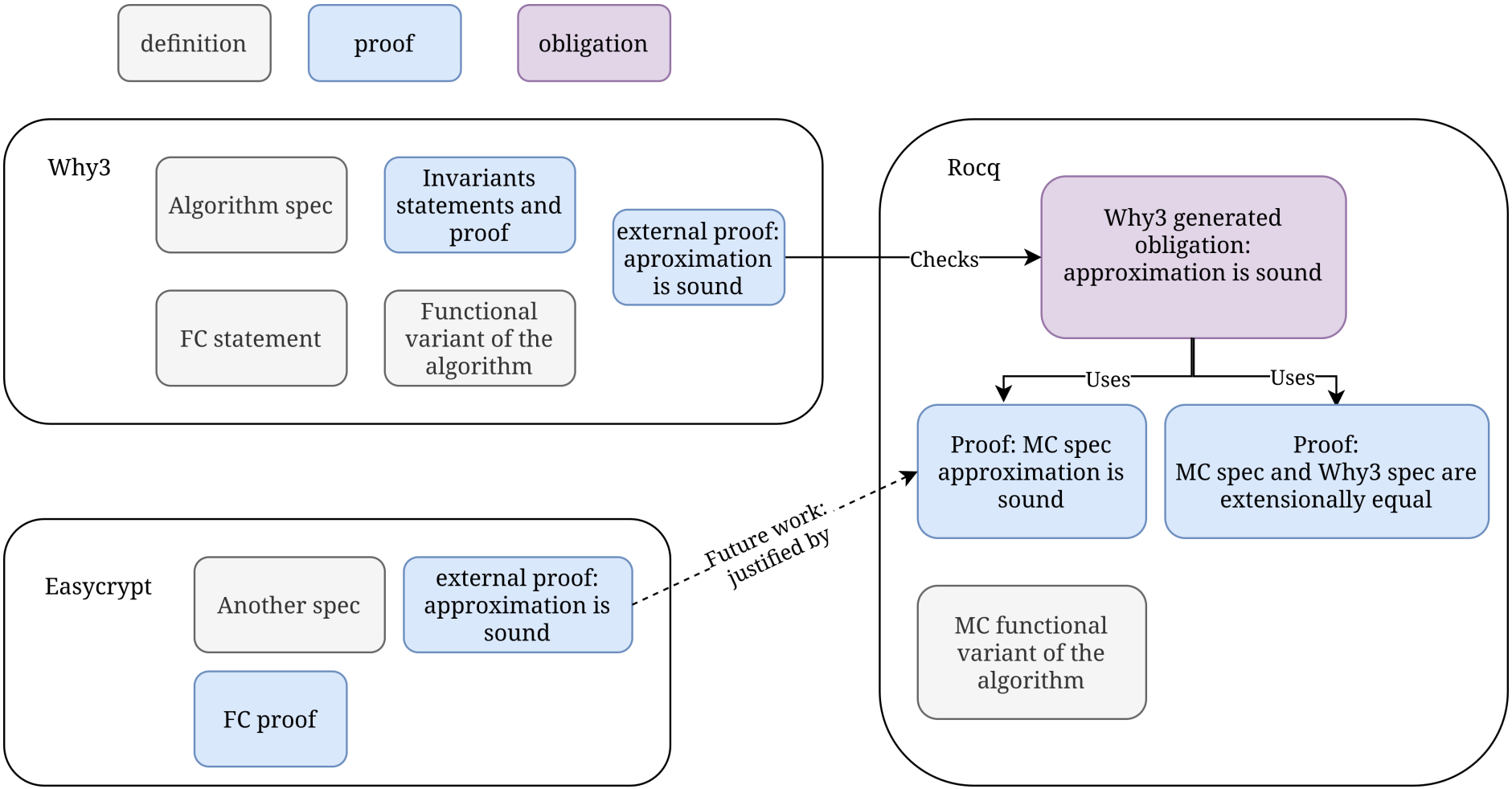
What is Why3?

- A platform for deductive program verification
- Language for specification and programming: WhyML
  - Polymorphic types, sum type, records with mutable fields
  - Reference to the value of variables at a point in the past
  - Ghost code
- Automated and interactive discharge of obligations
- Program extraction

What is the trusted code base of Why3?

- The external provers used to verify a specification
- The transformation from specifications into formulas for SAT solvers

Mechanization overview



Third difficulty: lack of effective tools. Part 1

Example: proving the grayed box.

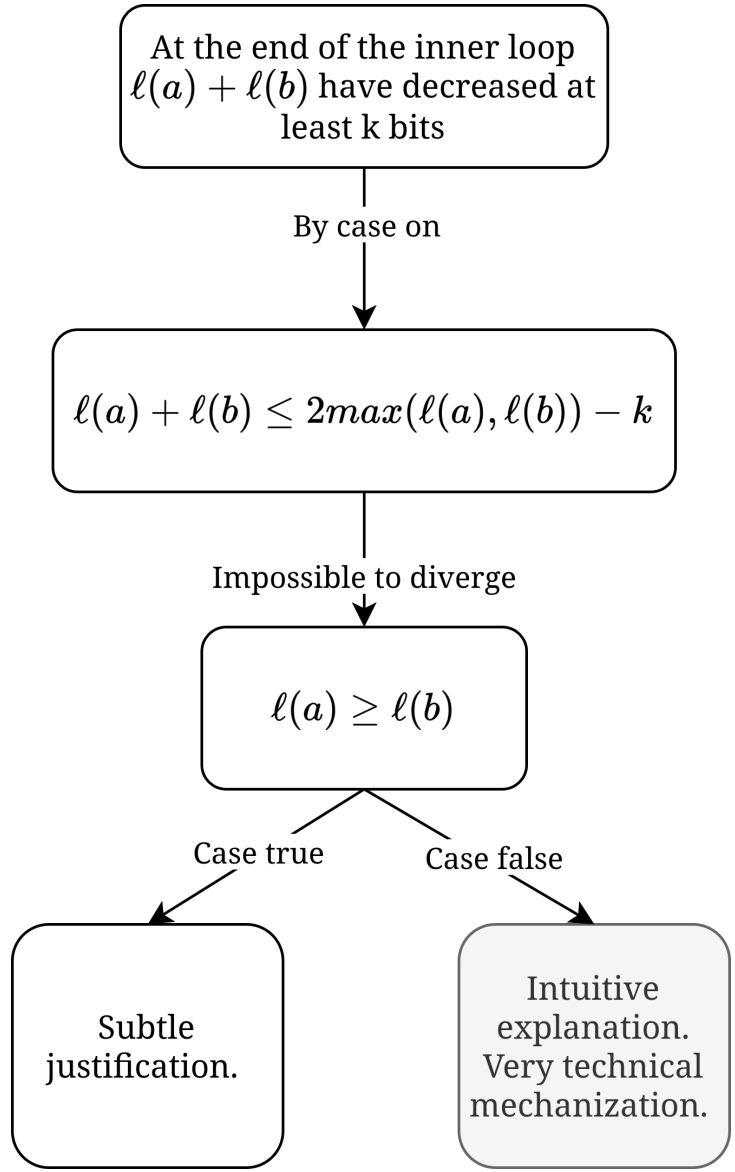
Proof strategy: it is exactly the box on the left after loops that halve followed by a swap.

Requires

- Double step induction
- Tons of transferring from $\mathbb{N}$ to $\mathbb{Z}$

Result

Two 200 LOC scripts.



At every optimization pass the number of:

- Ghost variables grow by 2-4, following the non optimized values
- Invariants grow by 10-15, proving relations between the optimization and non optimized values

№ of	GCD	(1)	(2)	(3)
invariants	6	11	37	~70
VCs	~100	~150	~300	~850
lemmas	15	25	55	236
Why3 interactive proofs	6	7	10	66

Final number of outer invariants: ~10.
Final number of inner invariants: ~60.

The outcome

- Why3 development ~2.9K LOC
- Divergence case proof in Rocq ~4.1K LOC (ssr)
- Obligation file obligation file: automatically generated ~3.6 LOC + ~700 LOC proof.

Proof delegation

Near zero-effort spent proving MC and Why3 specs were extensionally equal.

A strong post-condition genereator

Some invariants were not affected by specific branch

Ghost variables

Drove the invariant formalization blueprint

Automation
Some automated help with tactics like lia.

Induction
The right tool to do induction

Functional correctness of an optimized modular inversion algorithm

Tomás Vallejos Parada[1]
joint work with
Assia Mahboubi[1,2] Guillaume Melquiond[1] Pierre-Yves Strub[3]

Inria[1], France

Vrije Universiteit Amsterdam[2], The Netherlands

PQShield[3], France

April 9, 2026