Functional correctness of an optimized modular inversion algorithm

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

[1]Inria, France
[2]Vrije Universiteit Amsterdam, The Netherlands
[3]PQShield, France

2026-07-26 @ ITP'26 The talk will be uploaded soon @ tvallejos.cl

Context

Modular inversion

> For a prime m and $y \neq 0$, compute y^{-1}, such that, $yy^{-1} = 1 (\mathrm{mod}\, m)$.

Intensively used in cryptographic algorithm families, e.g. RSA and ECC

- **Efficiency** matters: large m (e.g. $2^{255} - 19$)
- **Security** constraints: constant time

In 2020 Pornin publishes an algorithm with ~25% perf gain w.r.t. best implem Bernstein and Yang (2019).
Used in:

1. **Falcon**: Post-quantum signature scheme (cf NIST PQ Cryptography Standardization)
2. **blst**: Library for performant signature schemes

Modular inversion via EGCD

```
mod_inv(y, m) :=
a <- y, b <- m, u <- 1, v <- 0
while a ≠ 0
    quotient <- b div a
    (b, a) <- (a, b - quotient * a)
    (v, u) <- (u, v - quotient * u)
return v
```

$$yy^{-1} = 1 (\mathrm{mod}\, m)$$

$$yy^{-1} + mu = 1 (\mathrm{mod}\, m)$$

$$yy^{-1} + mu = \gcd(y, m)$$

(Bezout's identity!)

A highly optimized modular inversion via EGCD

Algorithm 2 Extended Binary GCD (optimized algorithm)

Require: Odd modulus m ($m \geq 3$, $m \bmod 2 = 1$), value y ($0 \leq y < m$), and $k > 1$
Ensure: $1/y \bmod m$ (if $\mathrm{GCD}(y, m) = 1$)

$a \leftarrow y, u \leftarrow 1, b \leftarrow m, v \leftarrow 0$
for $1 \leq i \leq \lceil (2\mathrm{len}(m) - 1)/k \rceil$ **do**
  $n \leftarrow \max(\mathrm{len}(a), \mathrm{len}(b), 2k)$
  $\bar{a} \leftarrow (a \bmod 2^{k-1}) + 2^{k-1}\lfloor a/2^{n-k-1} \rfloor$ ▷ $\bar{a} < 2^{2k}$
  $\bar{b} \leftarrow (b \bmod 2^{k-1}) + 2^{k-1}\lfloor b/2^{n-k-1} \rfloor$ ▷ $\bar{b} < 2^{2k}$
  $f_0 \leftarrow 1, g_0 \leftarrow 0, f_1 \leftarrow 0, g_1 \leftarrow 1$
  for $1 \leq j \leq k - 1$ **do**
    if $\bar{a} = 0 \bmod 2$ **then**
      $(f_0, g_0) \leftarrow (f_0 - f_1, g_0 - g_1)$
    $(f_1, g_1) \leftarrow (2f_1, 2g_1)$
  $(a, b) \leftarrow ((af_0 + bg_0)/2^{k-1}, (af_1 + bg_1)/2^{k-1})$
  if $a < 0$ **then**
    $(a, f_0, g_0) \leftarrow (-a, -f_0, -g_0)$
  if $b < 0$ **then**
    $(b, f_1, g_1) \leftarrow (-b, -f_1, -g_1)$
  $(u, v) \leftarrow (uf_0 + vg_0 \bmod m, uf_1 + vg_1 \bmod m)$
$v \leftarrow v/2^{(k-1)\lceil (2\mathrm{len}(m)-1)/(k-1) \rceil} \bmod m$
if $b \neq 1$ **then**
  return 0 (value y is not invertible)

Note. An initial version of this note used a slightly more aggressive algorithm ( ) which was wrong.

Binary EGCD

```
bin_mod_inv(y, m   ) :=
a <- y, b <- m, u <- 1, v <- 0,
while a ≠ 0
    if a = 0 mod 2
        a <- a / 2
        u <- u / 2 mod m
    else
        if a < b
            swap(a,b)
            swap(u,v)
        a <- (a - b) / 2
        u <- (u - v) / 2 mod m
return v
```

Observe

- No decision is made on values `u` and `v`
- Parity needs only lowest bit
- `a < b` needs a sensible amount of higher bits

Possible optimizations?

- Do batch update of `u` and `v`
- Compute on approximations of `a` and `b`

Binary EGCD

```
bin_mod_inv(y, m   ) :=
a ← y, b ← m, u ← 1, v ← 0,
while a ≠ 0
    if a = 0 mod 2
        a ← a / 2
        u ← u / 2 mod m
    else
        if a < b
            swap(a,b)
            swap(u,v)
        a ← (a - b) / 2
        u ← (u - v) / 2 mod m
return v
```

Observe

- No decision is made on values `u` and `v`
- Parity needs only lowest bit
- `a < b` needs a sensible amount of higher bits

Possible optimizations?

- Do batch update of `u` and `v`
- Compute on approximations of `a` and `b`

Batch update of u and v

```
opt_mod_inv (y, m, k) :=
a <- y, b <- m, u <- 1, v <- 0          // init variables and coeff for a and b
while a ≠ 0
```

Batch update of u and v

```
opt_mod_inv (y, m, k) :=
a ← y, b ← m, u ← 1, v ← 0          // init variables and coeff for a and b
while a ≠ 0
    R                <- init_record()// records updates during execution
```

Batch update of u and v

```
opt_mod_inv (y, m, k) :=
a ← y, b ← m, u ← 1, v ← 0        // init variables and coeff for a and b
while a ≠ 0
    R              ← init_record()// records updates during execution

    do k iters
```

Batch update of u and v

```
opt_mod_inv (y, m, k) :=
a ← y, b ← m, u ← 1, v ← 0          // init variables and coeff for a and b
while a ≠ 0
    R                ← init_record()// records updates during execution

    do k iters
      if a = 0 mod 2
          a <- a / 2          ; R <- record_halving_a(R)
```

Batch update of u and v

```
opt_mod_inv (y, m, k) :=
a ← y, b ← m, u ← 1, v ← 0          // init variables and coeff for a and b
while a ≠ 0
    R               ← init_record()// records updates during execution

    do k iters
      if a = 0 mod 2
          a ← a / 2          ; R ← record_halving_a(R)
      else
          if a < b then
            swap(a,b)        ; R <- record_swap(R)
          a <- (a - b) / 2; R <- record_subtraction(R)
```

Batch update of u and v

```
opt_mod_inv (y, m, k) :=
a ← y, b ← m, u ← 1, v ← 0        // init variables and coeff for a and b
while a ≠ 0
    R              ← init_record()// records updates during execution

    do k iters
      if a = 0 mod 2
          a ← a / 2       ; R ← record_halving_a(R)
      else
          if a < b then
            swap(a,b)     ; R ← record_swap(R)
          a ← (a - b) / 2; R ← record_subtraction(R)

    (u,v)          <- batch_update(u,v,R,k) mod m
```

Batch update of u and v

```
opt_mod_inv (y, m, k) :=
a <- y, b <- m, u <- 1, v <- 0        // init variables and coeff for a and b
while a ≠ 0
    R              <- init_record()// records updates during execution

    do k iters
      if a = 0 mod 2
          a <- a / 2        ; R <- record_halving_a(R)
      else
          if a < b then
            swap(a,b)       ; R <- record_swap(R)
          a <- (a - b) / 2; R <- record_subtraction(R)

    (u,v)          <- batch_update(u,v,R,k) mod m
return v
```

Approximation-based optimization

Approximation based modular inversion

```
opt_mod_inv (y, m, k) :=
a <- y, b <- m, u <- 1, v <- 0        // init variables and coeff for a and b
while a ≠ 0
    R              <- init_record()// records updates during execution

    do k iters

                          ; R <- record_halving_a(R)


                          ; R <- record_swap(R)
                          ; R <- record_subtraction(R)


    (u,v)          <- batch_update(u,v,R,k) mod m
return v
```

Approximation based modular inversion

```
opt_mod_inv (y, m, k) :=
a <- y, b <- m, u <- 1, v <- 0        // init variables and coeff for a and b
while a ≠ 0
    R              <- init_record()// records updates during execution
    (α,β)          <- approx(a,b,k)
    do k iters

                           ; R <- record_halving_a(R)


                           ; R <- record_swap(R)
                           ; R <- record_subtraction(R)

    (a,b)          <- batch_update(a,b,R,k)
    (u,v)          <- batch_update(u,v,R,k) mod m
return v
```

Approximation based modular inversion

```
opt_mod_inv (y, m, k) :=
a <- y, b <- m, u <- 1, v <- 0        // init variables and coeff for a and b
while a ≠ 0
    R              <- init_record()// records updates during execution
    (α,β)          <- approx(a,b,k)
    do k iters
      if α = 0 mod 2
          α <- α / 2          ; R <- record_halving_a(R)
      else
          if α < β  then
              swap(α,β)       ; R <- record_swap(R)
          α <- (α - β) / 2    ; R <- record_subtraction(R)
    (a,b)          <- batch_update(a,b,R,k)
    (u,v)          <- batch_update(u,v,R,k) mod m
return v
```

Desynchronization

(1) have a, α odd, and $\alpha \geqslant \beta$, but $a < b$.
(2) have `α' := (α - β)/2` , and `a' := (a-b)/2` .
(3) have `α'` , and `a'` odd, and `α' < β`

```
if α = 0 mod 2
    α <- α / 2
else
    if α < β
        swap(α,β)
    α <- (α - β) / 2
```

Desynchronization

(1) have a, α odd, and $\alpha \geqslant \beta$, but $a < b$.
(2) have `α' := (α - β)/2` , and `a' := (a-b)/2` .
(3) have `α'` , and `a'` odd, and `α' < β`

```
if α = 0 mod 2
    α <- α / 2
else
    if α < β
        swap(α,β)
    α <- (α - β) / 2
```

Desynchronization

(1) have a, α odd, and $\alpha \geqslant \beta$, but $a < b$.
(2) have α' := (α - β)/2 , and a' := (a-b)/2 .
(3) have α', and a' odd, and α' < β

```
if α = 0 mod 2
    α <- α / 2
else
    if α < β
        swap(α,β)
    α <- (α - β) / 2
```

α is odd (else branch)

$\alpha \geq \beta$

$\alpha \leftarrow \alpha'$

Desynchronization

(1) have a, α odd, and $\alpha \geqslant \beta$, but $a < b$.
(2) have `α' := (α - β)/2` , and `a' := (a-b)/2` .
(3) have `α'`, and `a'` odd, and `α' < β`

```
if α = 0 mod 2
    α <- α / 2
else
    if α < β
        swap(α,β)
    α <- (α - β) / 2
```

α is odd (else branch)

$\alpha \geq \beta$

$\alpha \leftarrow \alpha'$

a is odd

$a < b$

$a \leftarrow a'$ ($\mathbf{a} < \mathbf{0}$!!)

Desynchronization

(1) have a, α odd, and $\alpha \geqslant \beta$, but $a < b$.

(2) have `α' := (α - β)/2` , and `a' := (a-b)/2` .

(3) have `α'`, and `a'` odd, and `α' < β`

```
if α = 0 mod 2
    α <- α / 2
else
    if α < β
        swap(α,β)
    α <- (α - β) / 2
```

α is odd (else branch)

$\alpha \geq \beta$

$\alpha \leftarrow \alpha'$

a is odd

$a < b$

$a \leftarrow a'$ ($a < 0$!!)

Desynchronization

(1) have a, α odd, and $\alpha \geqslant \beta$, but $a < b$.

(2) have `α' := (α - β)/2` , and `a' := (a-b)/2` .

(3) have `α'`, and `a'` odd, and `α' < β`

```
if α = 0 mod 2
    α <- α / 2
else
    if α < β
        swap(α,β)
    α <- (α - β) / 2
```

α is odd (else branch)

$\alpha \geq \beta$

$\alpha \leftarrow \alpha'$

α' is odd

$\alpha < \beta$ (swap)

$\alpha \leftarrow (\alpha - \beta)/2$

a is odd

$a < b$

$a \leftarrow a'$ ($a < 0$!!)

Desynchronization

(1) have a, α odd, and $\alpha \geqslant \beta$, but $a < b$.

(2) have `α' := (α - β)/2` , and `a' := (a-b)/2` .

(3) have `α'`, and `a'` odd, and `α' < β`

```
if α = 0 mod 2
    α <- α / 2
else
    if α < β
        swap(α,β)
    α <- (α - β) / 2
```

α is odd (else branch)

$\alpha \geq \beta$

$\alpha \leftarrow \alpha'$

`α'` is odd

$\alpha < \beta$ (swap)

`α ← (α - β)/2`

a is odd

$a < b$

a ← a' (**a < 0 !!**)

`a'` is odd

`a' < b` (**sync**)

`a'' ← (b - a')/2`

Desynchronization

(1) have a, α odd, and $\alpha \geqslant \beta$, but $a < b$.

(2) have `α' := (α - β)/2` , and `a' := (a-b)/2` .

(3) have `α'`, and `a'` odd, and `α' < β`

```
if α = 0 mod 2
    α <- α / 2
else
    if α < β
        swap(α,β)
    α <- (α - β) / 2
```

α is odd (else branch)

$\alpha \geq \beta$

$\alpha \leftarrow \alpha'$

`α'` is odd

$\alpha < \beta$ (swap)

$\alpha \leftarrow$ `(α - β)/2`

a is odd

$a < b$

$a \leftarrow a'$ ($a < 0$!!)

`a'` is odd

`a'` < b (**sync**)

`a'' ← (b - a')/2`

but `a' < 0`

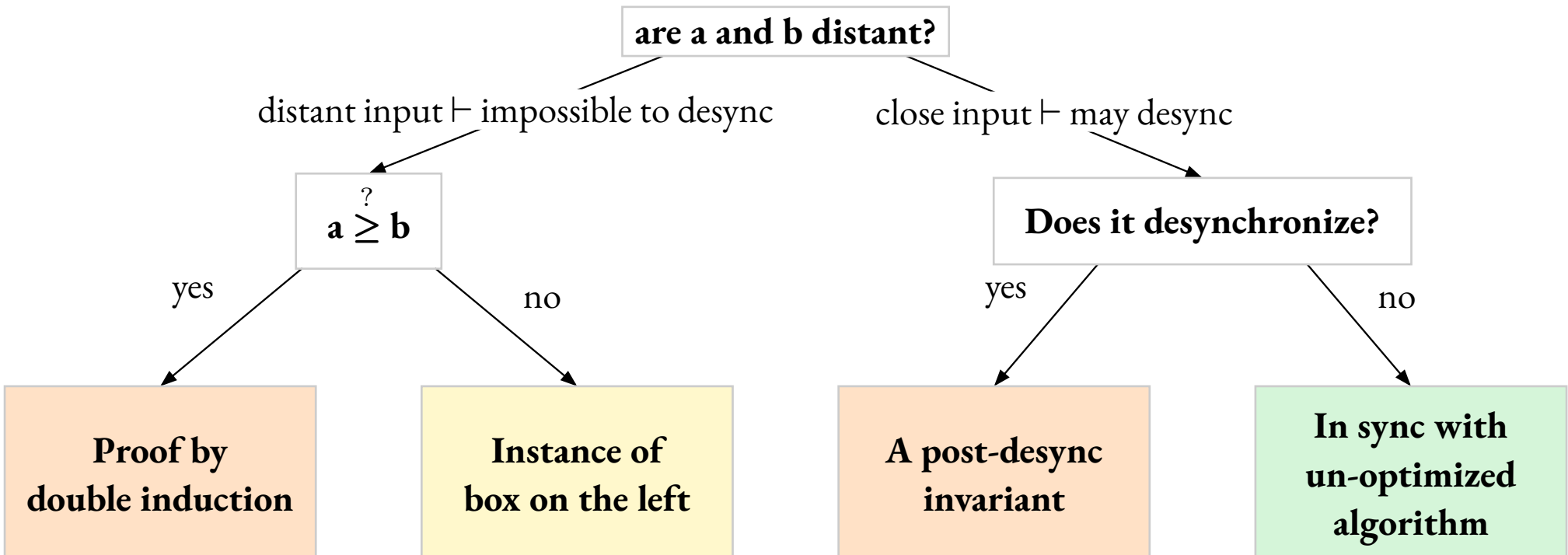
`a'' = (b + |a'|)/2`

`a' < b + a'`

Functional correctness

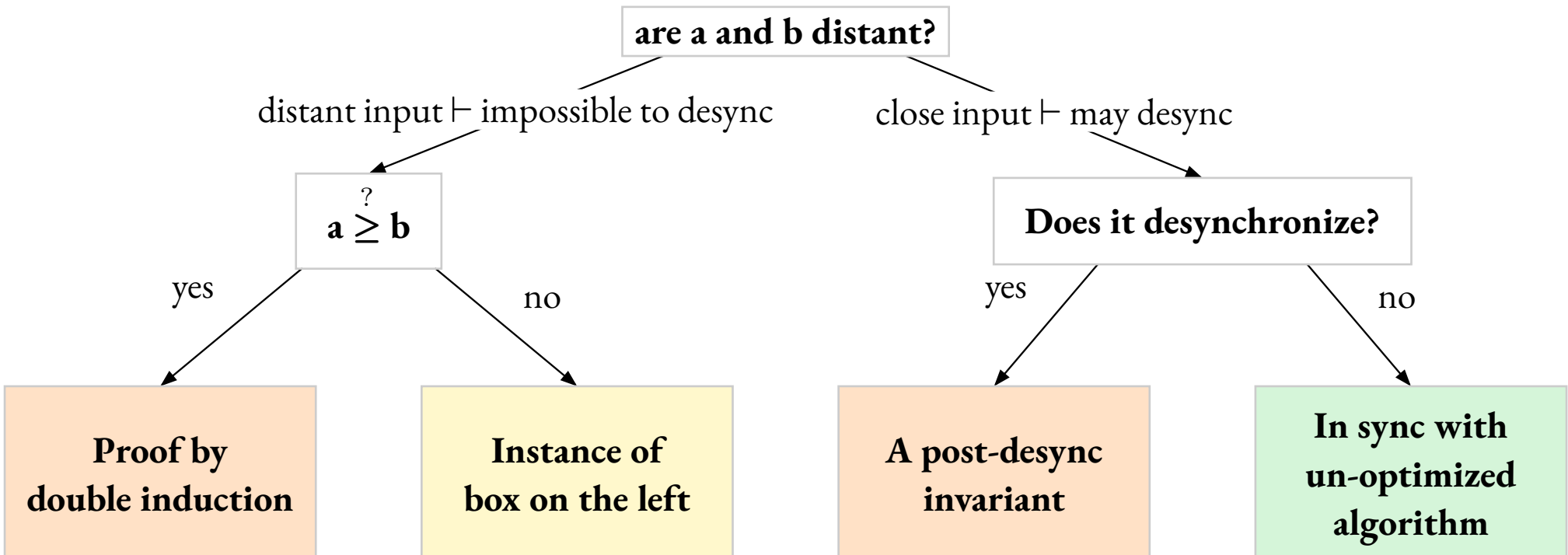
Why is using approximations correct?

Goal: show `a` converges to `0`, but after desynchronization `a - b` may increase.
Strategy: establish a contract for the approximation loop. After k iterations at least k bits have been lost.



Why is using approximations correct?

Goal: show `a` converges to `0`, but after desynchronization `a - b` may increase.
Strategy: establish a contract for the approximation loop. After k iterations at least k bits have been lost.

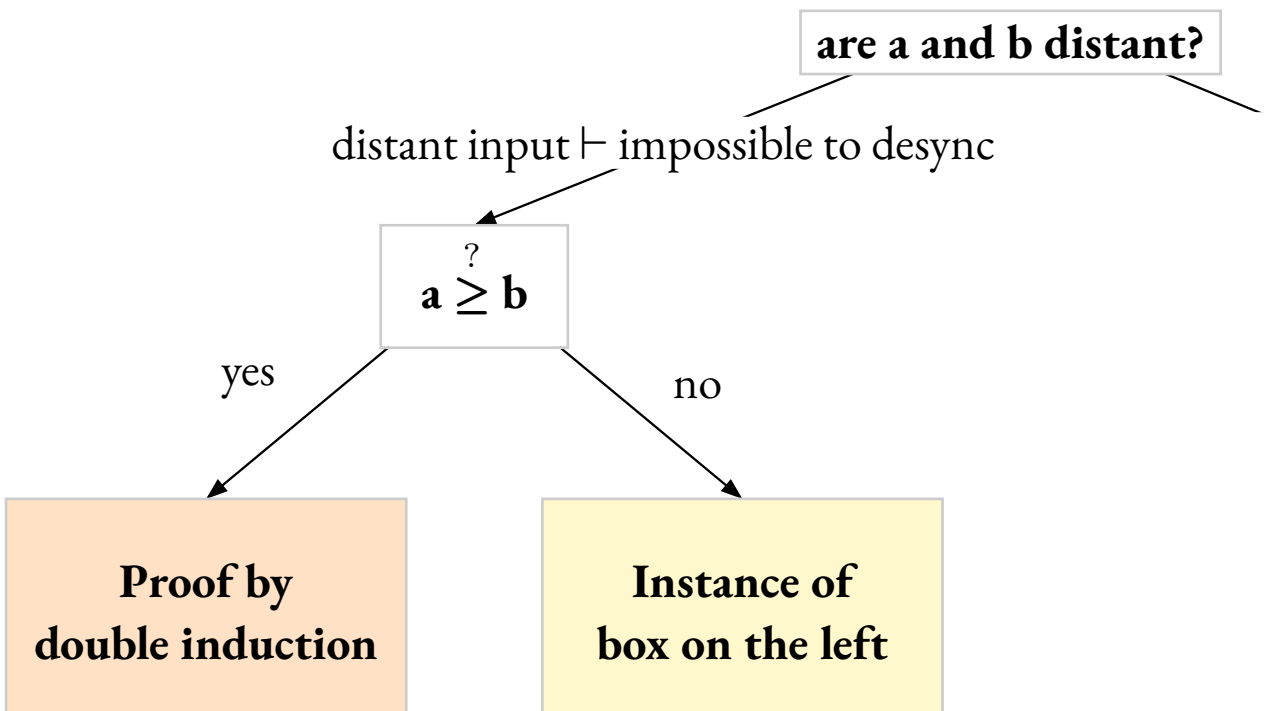


Why is using approximations correct?

Goal: show `a` converges to `0`, but after desynchronization `a - b` may increase.
Strategy: establish a contract for the approximation loop. After k iterations at least k bits have been lost.

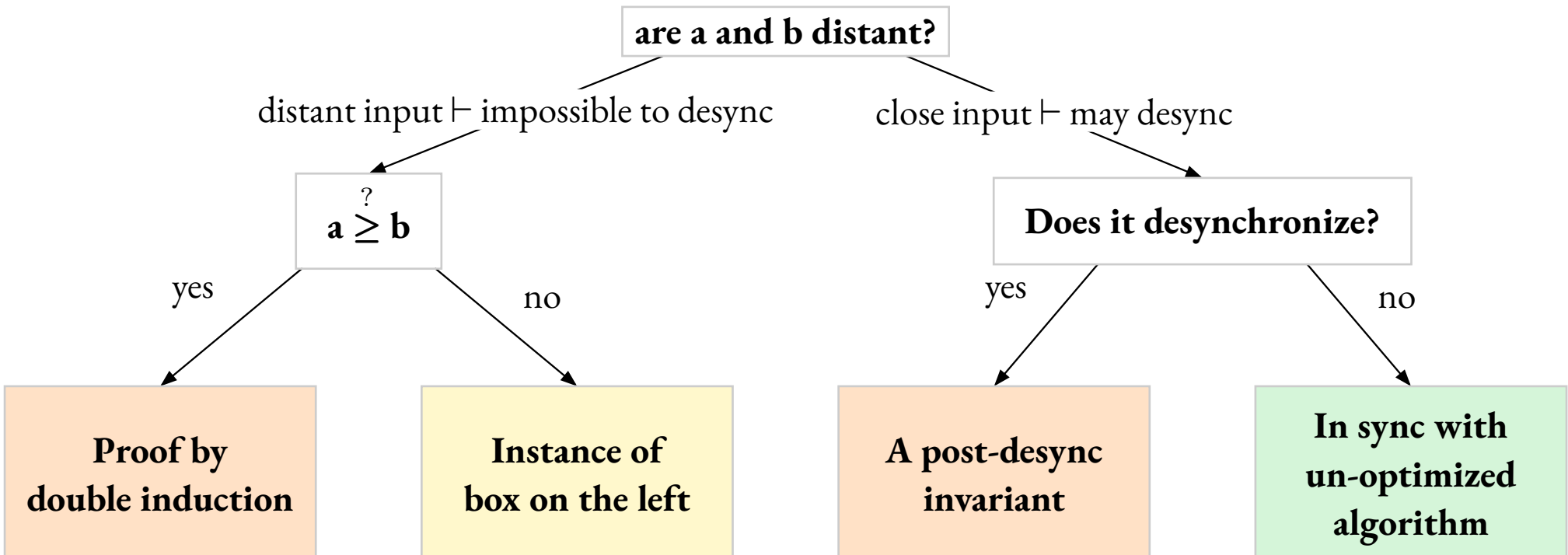
Why is using approximations correct?

Goal: show `a` converges to `0`, but after desynchronization `a - b` may increase.
Strategy: establish a contract for the approximation loop. After k iterations at least k bits have been lost.



Why is using approximations correct?

Goal: show `a` converges to `0`, but after desynchronization `a - b` may increase.
Strategy: establish a contract for the approximation loop. After k iterations at least k bits have been lost.



Desynchronization proof sketch

Let $\ell(x)$ be the length in bits of x.
Under hypotheses:

- $\ell(a)$ and $\ell(b)$ are close
- Desynchronization at iteration d

Then, after k iterations $\ell(a) + \ell(b)$ decreases at least k bits.

Proof sketch.

1. At iteration d, at least d bits were lost.
2. At iteration $d + 1$, k bits were lost.
3. $H(t) \triangleq P(t) \vee Q(t)$ is an invariant post-desync, and preserves k bits were lost.

With

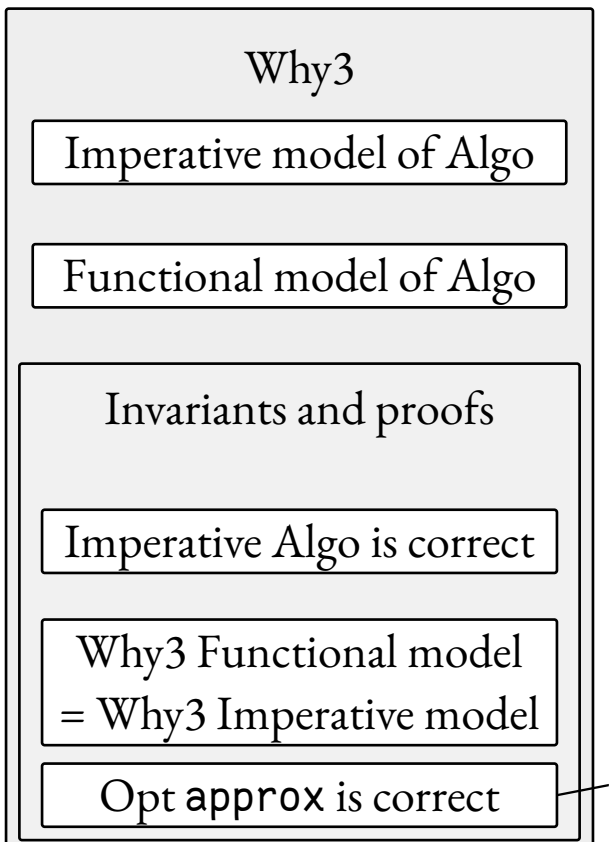
- $P(t) \triangleq M_t \geq 2^{n-k+1} \wedge -2^{n-k} < m_t \leq 0$
- $Q(t) \triangleq \max(|a_t|, |b_t|) < 2^{n-k+1}$
- $M_t \triangleq \max(a_t, b_t) \quad \wedge \quad m_d \triangleq \min(a_t, b_t) \quad a_t \triangleq$ `batch_update(a,b,R,t)` $\quad b_t \triangleq \ldots$

Mechanization

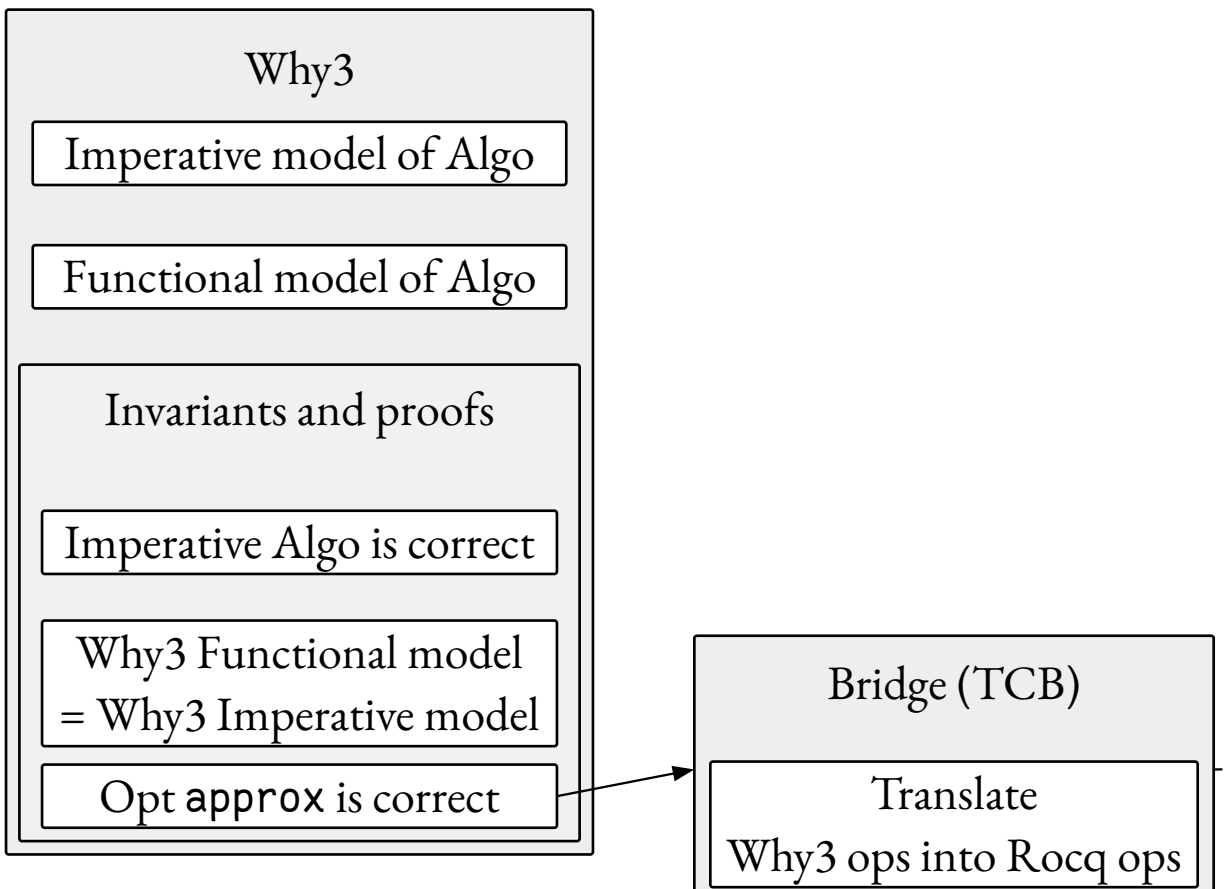
Collaboration between Why3 and Rocq

- Initial plan: a complete formalization in Why3
- Implement the binary EGCD algorithm in Why3
- Iterative refinement of optimizations/proofs
- At approximation optimization:
  - Context too big for Why3+SMT to handle
  - Lack of expressivity to state and prove theorems
- Rocq model and proof that approximation optimization is correct
- Connect the proof via a bridge between Why3 and Rocq

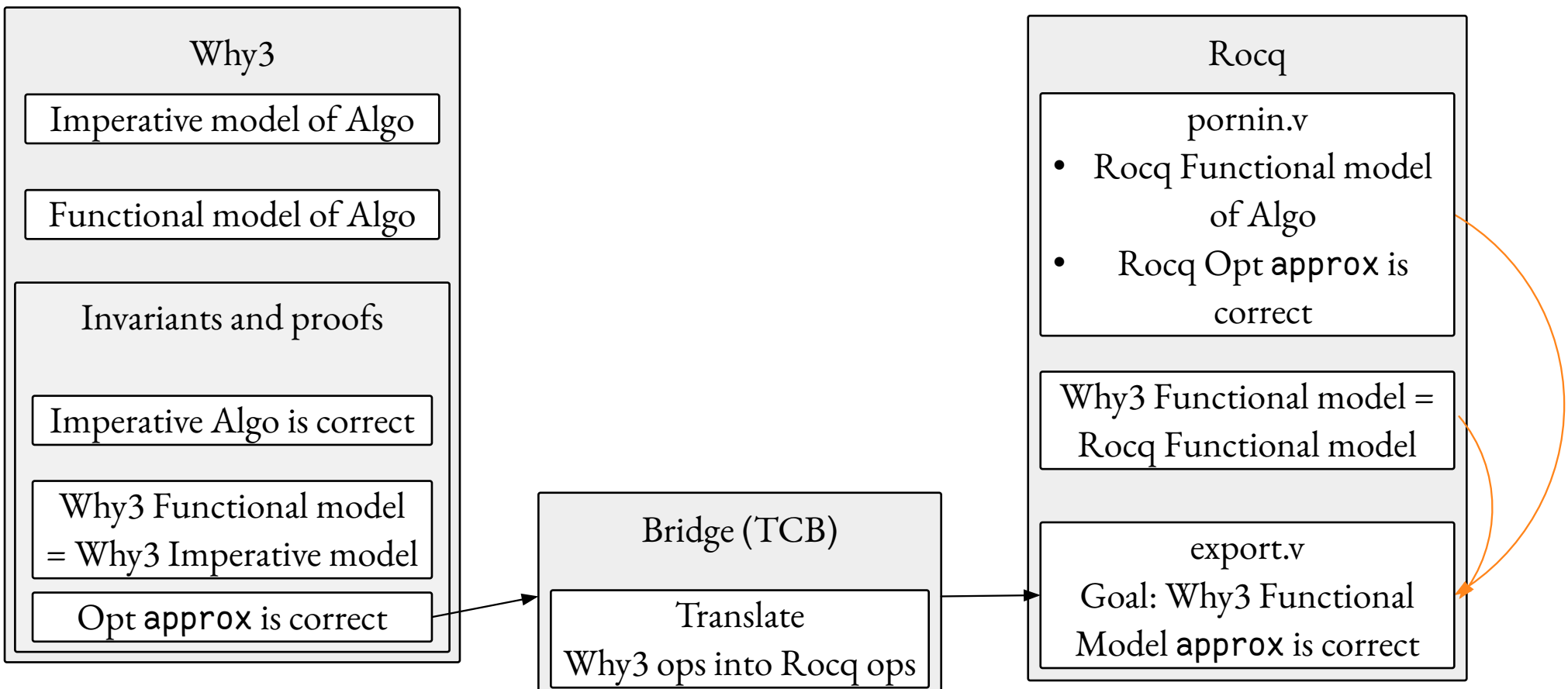
Overview of the mechanization



Overview of the mechanization



Overview of the mechanization



The bright side

- Near zero-effort spent glueing Rocq and Why3 models
- Some automation in Rocq for linear integer arithmetic
- No issue in mechanizing double induction arguments in Rocq

The not so bright side

- SMT solvers were not helpful with non-linear arithmetic
- Floyd-Hoare logic was not suitable to deal with desynchronization proofs
- Tons of proof transferring from $\mathbb{N}$ to $\mathbb{Z}$ (e.g. lift `e1 < e2` from `nat` to `int`)

Summary/Context

- Highly optimized algorithm used in critical security context
- Possibly incorrect functional correctness proof

Contributions

- Complete functional correctness proof
- Mechanization of the functional correctness proof
- Verification of an imperative implementation of the algorithm

The current version of the algorithm, and the revised proof, are believed correct.

Functional correctness of an optimized modular inversion algorithm

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

[1]Inria, France
[2]Vrije Universiteit Amsterdam, The Netherlands
[3]PQShield, France

2026-07-26 @ ITP'26 The talk will be uploaded soon @ tvallejos.cl