Formally Verified Montgomery Multiplication

Christoph Walther Orcid 0000−0002−9382−5399

Technische Universität Darmstadt, Darmstadt, Germany
Chr.Walther@informatik.tu-darmstadt.de

Abstract. We report on a machine assisted verification of an efficient implementation of Montgomery Multiplication which is a widely used method in cryptography for efficient computation of modular exponentiation. We shortly describe the method, give a brief survey of the `VeriFun` system used for verification, present the formal proofs and report on the effort for creating them. Our work uncovered a serious fault in a published algorithm for computing multiplicative inverses based on Newton-Raphson iteration, thus providing further evidence for the benefit of computer-aided verification.

Keywords Modular Arithmetic, Multiplicative Inverses, Montgomery Multiplication, Program Verification, Theorem Proving by Induction

1 Introduction

Montgomery Multiplication [6] is a method for efficient computation of residues $a^j \ mod \ n$ which are widely used in cryptography, e.g. for RSA, Diffie-Hellman, ElGamal, DSA, ECC etc. [4, 5]. The computation of these residues can be seen as an iterative calculation in the commutative ring with identity $R_n = (\mathbb{N}_n, \oplus, \mathsf{i}_n, \odot, 0, 1 \ mod \ n)$ where $n \geq 1$, $\mathbb{N}_n = \{0, \ldots, n-1\}$, addition defined by $a \oplus b = a + b \ mod \ n$, inverse operator defined by $\mathsf{i}_n(a) = a \cdot (n-1) \ mod \ n$, multiplication defined by $a \odot b = a \cdot b \ mod \ n$, neutral element 0 and identity 1 $mod \ n$.

For any $m \in \mathbb{N}$ relatively prime to n, some $m_n^{-1} \in \mathbb{N}_n$ exists such that $m \odot m_n^{-1} = 1 \ mod \ n$. m_n^{-1} is called the *multiplicative inverse* of m in R_n and is used to define a further commutative ring with identity $R_n^m = (\mathbb{N}_n, \oplus, \mathsf{i}_n, \otimes, 0, m \ mod \ n)$ where multiplication is defined by $a \otimes b = a \odot b \odot m_n^{-1}$ and identity given as $m \ mod \ n$. The multiplication $\otimes$ of R_n^m is called *Montgomery Multiplication*.

The rings R_n and R_n^m are isomorphic by the isomorphism $h : R_n \to R_n^m$ defined by $h(a) = a \odot m$ and $h^{-1} : R_n^m \to R_n$ given by $h^{-1}(a) = a \odot m_n^{-1}$. Consequently $a \cdot b \ mod \ n$ can be calculated in ring R_n^m as well because

$$a \cdot b \ mod \ n = a \odot b = h^{-1}(h(a \odot b)) = h^{-1}(h(a) \otimes h(b)) \ . \qquad (*)$$

The required operations $h, \otimes$ and h^{-1} can be implemented by the so-called *Montgomery Reduction* redc [6] (displayed in Fig. 1) as stated by Theorem 1:

```
function redc(x, z, m, n : ℕ) : ℕ <=
if m ≠ 0
 then let q := (x + n · (x · z mod m))/m in
        if n > q then q else q − n end_if
      end_let
end_if

function redc*(x, z, m, n, j : ℕ) : ℕ <=
if m ≠ 0
  then if n ≠ 0
         then if j = 0
                then m mod n
                else redc(x · redc*(x, z, m, n, ¯(j)), z, m, n)
              end_if
          end_if
end_if
```

Fig. 1. Procedures redc and redc* implementing the Montgomery Reduction

Theorem 1. *Let $a, b, n, m \in \mathbb{N}$ with $m > n > a$, $n > b$ and n, m relatively prime, let $I = \mathfrak{i}_m(n_m^{-1})$ and let $M = m^2$ mod n. Then I is called the* Montgomery Inverse *and (1) $h(a) = redc(a \cdot M, I, m, n)$, (2) $a \otimes b = redc(a \cdot b, I, m, n)$, and (3) $h^{-1}(a) = redc(a, I, m, n)$.*

By $(*)$ and Theorem 1, $a \cdot b$ *mod* n can be computed by procedure redc and consequently a^j *mod* n can be computed by iterated calls of redc (implemented by procedure redc* of Fig. 1) as stated by Theorem 2:

Theorem 2. *Let a, n, m, I and M like in Theorem 1. Then for all $j \in \mathbb{N}$:*

$$a^j \ mod\ n = redc(redc^*(redc(a \cdot M, I, m, n), I, m, n, j), I, m, n) \ .^1$$

By Theorem 2, $j + 2$ calls of redc are required for computing a^j *mod* n, viz. one call to map a to $h(a)$, j calls for the Montgomery Multiplications and one call for mapping the result back with h^{-1}. This approach allows for an efficient computation of a^j *mod* n in R_n^m (for sufficient large j), if m is chosen as a power of 2 and some odd number for n, because x *mod* m then can be computed with constant time and x/m only needs an effort proportional to *log* m in procedure redc, thus saving the expensive *mod* n operations in R_n.

2 About ✓eriFun

The truth of Theorems 1 and 2 is not obvious at all, and some number theory with modular arithmetic is needed for proving them. Formal proofs are worthwhile because correctness of cryptographic methods is based on these theorems.

[1] Exponentiation is defined here with $0^0 = 1$ so that $redc(redc^*(redc(0 \cdot M, I, m, n), I, m, n, 0), I, m, n) = 1$ *mod* n holds in particular.

```
structure bool                          <=  true, false
structure ℕ                             <=  0, ⁺(⁻:ℕ)
structure signs                         <=  '+', '−'
structure ℤ                             <=  [outfix] ⟨ : ⟩(sign:signs, [outfix]| : ℕ)
structure triple[@T1, @T2, @T3]         <=  [outfix] ≪ : ≫ ( [postfix]₁ : @T1,
                                                [postfix]₂ : @T2, [postfix]₃ : @T3 )

lemma z ≠ 0 → [x · (y mod z) ≡ x · y] mod z <= ∀x, y, z:ℕ
if{¬ z = 0, (x · (y mod z) mod z) = (x · y mod z), true}
```

Fig. 2. Data structures and lemmas in ✓eriFun

Proof assistants like Isabelle/HOL, HOL Light, Coq, ACL2 and others have been shown successful for developing formal proofs in Number Theory (see e.g. [14]). Here we use the ✓eriFun system[2] [7, 10] to verify correctness of Montgomery Multiplication by proving Theorems 1 and 2. The system's object language consists of universal first-order formulas plus parametric polymorphism. Type variables may be instantiated with polymorphic types. Higher-order functions are not supported. The language provides principles for defining data structures, procedures operating on them, and for statements (called "lemmas") about the data structures and procedures. Unicode symbols may be used and function symbols can be written in out-, in-, pre- and postfix notation so that readability is increased by use of the familiar mathematical notation. Fig. 2 displays some examples. The data structure bool and the data structure $\mathbb{N}$ for natural numbers built with the constructors 0 and ⁺(...) for the successor function are the only predefined data structures in the system. ⁻(...) is the selector of ⁺(...) thus representing the predecessor function. Subsequently we need integers $\mathbb{Z}$ as well which we define in Fig. 2 as signed natural numbers. For instance, the expression $\langle\, '-', 42 \,\rangle$ is a data object of type $\mathbb{Z}$, selector *sign* yields the sign of an integer (like '−' in the example), and selector $|\ldots|$ gives the absolute value of an integer (like 42 in the example). Identifiers preceded by @ denote type variables, and therefore polymorphic triples are defined in Fig. 2. The expression $\ll 42, \langle\, '+', 47 \,\rangle, \langle\, '-', 5 \,\rangle \gg$ is an example of a data object of type *triple*$[\mathbb{N}, \mathbb{Z}, \mathbb{Z}]$. The i^{th} component of a triple is obtained by selector $(\ldots)_i$.

Procedures are defined by *if*- and *case*-conditionals, functional composition and recursion like displayed in Fig. 1. Procedure calls are evaluated eagerly, i.e. call-by-value. The use of incomplete conditionals like for redc and redc* results in incompletely defined procedures [12]. Such a feature is required when working with polymorphic data structures but is useful for monomorphic data structures too as it avoids the need for stipulating artificial results, e.g. for $n/0$. Predicates are defined by procedures with result type bool. Procedure function [infix] >(x, y:ℕ):bool <= ... for deciding the greater-than relation is the only prede-

[2] An acronym for "A Verifier for Functional Programs".

fined procedure in the system. Upon the definition of a procedure, ✓eriFun's automated termination analysis (based on the method of *Argument-Bounded Functions* [8, 11]) is invoked for generating termination hypotheses which are sufficient for the procedure's termination and proved like lemmas. Afterwards induction axioms are computed from the terminating procedures' recursion structure to be on stock for future use.

Lemmas are defined with conditionals $if : bool \times bool \times bool \to bool$ as the main connective, but negation $\neg$ and *case*-conditionals may be used as well. Only universal quantification is allowed for the variables of a lemma. Fig. 2 displays a lemma about (the elsewhere defined) procedure mod (computing the remainder function) which is frequently used in subsequent proofs. The string in the headline (between "lemma" and "<=") is just an identifier assigning a name to the lemma for reference and must not be confused with the statement of the lemma given as a boolean term in the lemma body. Some basic lemmas about equality and >, e.g. stating transitivity of = and >, are predefined in the system. Predefined lemmas are frequently used in almost every case study so that work is eased by having them always available instead of importing them from some proof library.

Lemmas are proved with the *HPL*-calculus (abbreviating *Hypotheses*, *Programs* and *Lemmas*) [10]. The most relevant proof rules of this calculus are *Induction*, *Use Lemma*, *Apply Equation*, *Unfold Procedure*, *Case Analysis* and *Simplification*. Formulas are given as sequents of form $H, IH \vdash goal$, where H is a finite set of *hypotheses* given as literals, i.e. negated or unnegated predicate calls and equations, IH is a finite set of *induction hypotheses* given as partially quantified boolean terms and *goal* is a boolean term, called the *goalterm* of the sequent. A deduction in the *HPL*-calculus is represented by a tree whose nodes are given by sequents. A lemma ℓ with body $\forall \ldots goal$ is *verified* iff *(i)* the goalterm of each sequent at a leaf of the proof tree rooted in $\{\}, \{\} \vdash goal$ equals *true* and *(ii)* each lemma applied by *Use Lemma* or *Apply Equation* when building the proof tree is *verified*. The base of this recursive definition is given by lemmas being proved without using other lemmas. Induction hypotheses are treated like *verified* lemmas, however being available only in the sequent they belong to.

The *Induction* rule creates the base and step cases for a lemma from an induction axiom. By choosing *Simplification*, the system's first-order theorem prover, called the *Symbolic Evaluator*, is started for rewriting a sequent's goalterm using the hypotheses and induction hypotheses of the sequent, the definitions of the data structures and procedures as well as the lemmas already *verified*. This reasoner is guided by heuristics, e.g. for deciding whether to use a procedure definition, for speeding up proof search by filtering out useless lemmas, etc. Equality reasoning is implemented by conditional term rewriting with AC-matching, where the orientation of equations is heuristically established [13]. The Symbolic Evaluator is a fully automatic tool over which the user has no control, thus leaving the *HPL*-proof rules as the only means to guide the system to a proof.

Also the *HPL*-calculus is controlled by heuristics. When applying the *Verify* command to a lemma, the system starts to compute a proof tree by choosing

appropriate *HPL*-proof rules heuristically. If a proof attempt gets stuck, the user must step in by applying a proof rule to some leaf of the proof tree (sometimes after pruning some unwanted branch of the tree), and the system then takes over control again. Also it may happen that a further lemma must be formulated by the user before the proof under consideration can be completed. All interactions are menu driven so that typing in proof scripts is avoided (see [7, 10]).

✓`eriFun` is implemented in JAVA and installers for running the system under *Windows*, *Unix/Linux* or *Mac* are available from the web [7]. When working with the system, we use proof libraries which had been set up over the years by extending them with definitions and lemmas being of general interest. When importing a definition or a lemma from a library into a case study, all program elements and proofs the imported item depends on are imported as well. The correctness proofs for Montgomery Multiplication depend on 9 procedures and 96 lemmas from our arithmetic proof library, which ranges from simple statements like associativity and commutativity of addition up to more ambitious theorems about primes and modular arithmetic. In the sequel we will only list the lemmas which are essential to understand the proofs and refer to [7] for a complete account of all used lemmas and their proofs.

3 Multiplicative Inverses

We start our development by stipulating how multiplicative inverses are computed. To this effect we have to define some procedure $\mathfrak{I} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ satisfying

$$\forall x, y{:}\mathbb{N}\ \ y \neq 0 \wedge gcd(x, y) = 1 \to [x \cdot \mathfrak{I}(x, y) \equiv 1]\ mod\ y \tag{1}$$

$$\forall x, y, z{:}\mathbb{N}\ \ y \neq 0 \wedge gcd(x, y) = 1 \to [z \cdot x \cdot \mathfrak{I}(x, y) \equiv z\,]\ mod\ y \tag{2}$$

$$\forall n, x, y, z{:}\mathbb{N}\ \ y \neq 0 \wedge gcd(x, y) = 1 \to [n + z \cdot x \cdot \mathfrak{I}(x, y) \equiv n + z\,]\ mod\ y.\ ^{3} \tag{3}$$

Lemma 2 is proved with Lemma 1 and library lemma

$$\forall n, m, x, y{:}\mathbb{N}\ \ gcd(n, m) = 1 \wedge [m \cdot x \equiv m \cdot y]\ mod\ n \to [x \equiv y]\ mod\ n \tag{4}$$

after instructing the system to use library lemma

$$\forall x, y, z{:}\mathbb{N}\ \ z \neq 0 \to [x \cdot (y\ mod\ z) \equiv x \cdot y\,]\ mod\ z \tag{5}$$

and ✓`eriFun` proves Lemma 3 automatically using Lemma 2 as well as library lemma

$$\forall n, x, y, z{:}\mathbb{N}\ \ z \neq 0 \wedge [x \equiv y]\ mod\ z \to [x + n \equiv y + n]\ mod\ z\ . \tag{6}$$

Multiplicative inverses can be computed straightforwardly with Euler's ϕ-function, where Lemma 1 then is proved with Euler's Theorem [7, 14]. But this approach is very costly and therefore unsuitable for an implementation of Montgomery Multiplication.

[3] If $x, y, z \in \mathbb{Z}$ and $n \in \mathbb{N}$, then $n \mid z$ abbreviates $z\ mod\ n = 0$, where $z\ mod\ n = -(|z|\ mod\ n)$ if $z < 0$, and $[x \equiv y]\ mod\ n$ stands for $n \mid x - y$. $x\ mod\ n = y\ mod\ n$ is sufficient for $[x \equiv y]\ mod\ n$ but only necessary, if x and y have same polarity.

```
function euclid(x, y:ℕ):triple[ℕ, ℤ, ℤ] <=
if y = 0
 then ⩽ x, ⟨'+', 1⟩, ⟨'+', 0⟩ ⩾
 else let e := euclid(y, (x mod y)), g := (e)1, s := (e)2, t := (e)3 in
        case sign(s) of
         '+' : ⩽ g, ⟨'−', |t|⟩, ⟨'+', |s| + (x/y) · |t|⟩ ⩾,
         '−' : ⩽ g, ⟨'+', |t|⟩, ⟨'−', |s| + (x/y) · |t|⟩ ⩾
        end_case
       end_let
end_if

function ℑB(x, y:ℕ):ℕ <=
if y ≠ 0
 then let s := (euclid(x, y))2 in
        case sign(s) of '+' : (|s| mod y), '−' : y − (|s| mod y) end_case
       end_let
end_if
```

Fig. 3. Computation of multiplicative inverses by the extended Euclidean algorithm

3.1 Bézout's Lemma

A more efficient implementation of procedure $\mathfrak{I}$ is based on Bézout's Lemma stating that the greatest common divisor can be represented as a linear combination of its arguments:

Bézout's Lemma
For all $x, y \in \mathbb{N}$ *some* $s, t \in \mathbb{Z}$ *exist such that* $gcd(x, y) = x \cdot s + y \cdot t$.

If $y \neq 0$, $\mathfrak{I}_B(x, y) := s \ mod \ y$ is defined and $gcd(x, y) = 1$ holds, then by Bézout's Lemma $[x \cdot \mathfrak{I}_B(x, y) = x \cdot (s \ mod \ y) \equiv x \cdot s \equiv x \cdot s + y \cdot t = 1] \ mod \ y$. To implement this approach, the integer s need to be computed which can be performed by the extended Euclidean algorithm displayed in Fig. 3. This approach is more efficient because a call of $euclid(x, y)$ (and in turn of $\mathfrak{I}_B(x, y)$ given as in Fig. 3) can be computed in time proportional to $(log \, y)^2$ if $x < y$, whereas the use of Euler's ϕ-function needs time proportional to $2^{log \, y}$ in the context of Montgomery Multiplication (as $\phi(2^{k+1}) = 2^k$).

However, $s \in \mathbb{Z}$ might be negative so that $y + (s \ mod \ y) \in \mathbb{N}$ instead of $s \ mod \ y$ then must be used as the multiplicative inverse of x because the carriers of the rings R_n and R_n^m are subsets of $\mathbb{N}$. We therefore define $\mathfrak{I}_B$ as shown in Fig. 3 which complicates the proof of Lemma 1 (with $\mathfrak{I}$ replaced by $\mathfrak{I}_B$) as this definition necessitates a proof of $[x \cdot y + x \cdot (s \ mod \ y) \equiv 1] \ mod \ y$ if $s < 0$.

Bézout's Lemma is formulated in our system's notation by the pair of lemmas displayed in Fig. 4. When prompted to prove Lemma 7, the system starts a Peano induction upon x but gets stuck in the step case. We therefore command to use induction corresponding to the recursion structure of procedure euclid. ✓veriFun

```
lemma Bézout's Lemma #1 <= ∀ x, y : ℕ
let e := euclid(x, y), g := (e)1, s := (e)2, t := (e)3 in
 case sign(s) of '+' : x·|s| = y·|t| + g, '−' : x·|s| + g = y·|t| end_case
end_let                                                                    (7)

lemma Bézout's Lemma #2 <= ∀ x, y : ℕ (euclid(x, y))1 = gcd(x, y) .        (8)
```

Fig. 4. Bézout's Lemma

responds by proving the base case and simplifying the induction conclusion in case $sign(s) =$ '+' to

$$y \neq 0 \rightarrow x \cdot |t| + g = (x \ mod \ y) \cdot |t| + g + |t| \cdot (y-1) \cdot (x/y) + |t| \cdot (x/y) \quad \text{(i)}$$

(where e abbreviates $euclid(y, x \ mod \ y)$, $g := (e)_1$, $s := (e)_2$ and $t := (e)_3$) using the induction hypothesis

$$\begin{array}{l} \forall x'{:}\mathbb{N}\ let\{e := euclid(x', (x \ mod \ y),\ g := (e)_1,\ s := (e)_2,\ t := (e)_3; \\ \quad case\{sign(s); \\ \quad\quad \text{'+'}:\ x' \cdot |\,s\,| = (x \ mod \ y) \cdot |\,t\,| + g, \\ \quad\quad \text{'−'}:\ x' \cdot |\,s\,| + g = (x \ mod \ y) \cdot |\,t\,|\}\} \end{array}$$

and some basic arithmetic properties. We then instruct the system to use the quotient-remainder theorem for replacing x at the left-hand side of the equation in (i) by $(x/y) \cdot y + (x \ mod \ y)$ causing ✓eriFun to complete the proof. The system computes a similar proof obligation for case $sign(s) =$ '−' which is proved in the same way.

By "basic arithmetic properties" we mean well known facts like associativity, commutativity, distributivity, cancellation properties etc. of $+, -, \cdot, /, gcd, \ldots$ which are defined and proved in our arithmetic proof library. These facts are used almost everywhere by the Symbolic Evaluator so that we will not mention their use explicitly in the sequel.

When called to prove Lemma 8 by induction corresponding to the recursion structure of procedure euclid, ✓eriFun responds by proving the base case and rewrites the step case with the induction hypothesis to

$$y \neq 0 \rightarrow gcd(x, y) = gcd(y, (x \ mod \ y)) \ . \quad \text{(ii)}$$

It then automatically continues with proving (ii) by induction corresponding to the recursion structure of procedure gcd where it succeeds for the base and the step case. Lemma 8 is useful because it relates procedure euclid to procedure gcd of our arithmetic proof library so that all lemmas about gcd can be utilized for the current proofs.

For proving the inverse property

$$\forall x, y{:}\mathbb{N}\ \ y \neq 0 \wedge gcd(x, y) = 1 \rightarrow [x \cdot \mathfrak{I}_B(x, y) \equiv 1] \ mod \ y \quad (9)$$

of procedure $\mathfrak{I}_B$, we call the system to unfold procedure call $\mathfrak{I}_B(x, y)$. ✓eriFun responds by proving the statement for case $sign(s) =$ '+' using Bézout's Lemma 7 and 8 and the library lemmas

$$\forall x, y, z{:}\mathbb{N}\ \ z \neq 0 \wedge z \mid x \rightarrow [x + y \equiv y]\ mod\ z \tag{10}$$

$$\forall x, y{:}\mathbb{N}\ \ y \neq 0 \rightarrow y \mid x \cdot y \tag{11}$$

as well as (5), but gets stuck in the remaining case with proof obligation

$$y \neq 0 \wedge sign(s) = \text{'}-\text{'} \wedge g = 1 \rightarrow [x \cdot y - x \cdot (|s|\ mod\ y) \equiv 1]\ mod\ y \tag{iii}$$

where g abbreviates $(euclid(x, y))_1$ and s stands for $(euclid(x, y))_2$. Proof obligation (iii) represents the unpleasant case of the proof development and necessitates the invention of an auxiliary lemma for completing the proof. After some unsuccessful attempts, we eventually came up with lemma

$$\forall x, y, z, u{:}\mathbb{N}\ y \neq 0 \wedge y \mid (x \cdot z + u) \wedge x \geq u \rightarrow [x \cdot y - x \cdot (z\ mod\ y) \equiv u]\ mod\ y\ . \tag{12}$$

For proving (iii), we command to use Lemma 12 for replacing the left-hand side of the congruence in (iii) by g, and ✓eriFun computes

$$\begin{array}{l} y \neq 0 \wedge sign(s) = \text{'}-\text{'} \wedge g = 1 \rightarrow \\ \quad (x \geq g \rightarrow y \mid (x \cdot |s| + g)) \wedge \\ \quad (x < g \rightarrow [x \cdot y - x \cdot (|s|\ mod\ y) \equiv 1]\ mod\ y\ . \end{array} \tag{iv}$$

Now we can call the system to use Bézout's Lemma 7 for replacing $x \cdot |s| + g$ in (iv) by $y \cdot |t|$ causing ✓eriFun to complete the proof with Bézout's Lemma 8 and library lemma (11) in case of $x \geq g$ and otherwise showing that $x < g = 1$ entails $x = 0$ and $1 = g = gcd(0, y) = y$ in turn, so that $x \cdot y - x \cdot (|s|\ mod\ y)$ simplifies to 0 and $[0 \equiv 1]\ mod\ y$ rewrites to *true*.

It remains to prove auxiliary lemma (12) for completing the proof of Lemma 9: After being called to use library lemma

$$\forall x, y, z{:}\mathbb{N}\ \ z \neq 0 \wedge z \mid (x - y) \wedge z \mid (y - x) \rightarrow [x \equiv y\,]\ mod\ z\ ^{4} \tag{13}$$

for replacing the left-hand side of the congruence in (12) by u, ✓eriFun computes

$$y \neq 0 \wedge y \mid (x \cdot z + u) \wedge x \geq u \rightarrow y \mid (u - (x \cdot y - x \cdot (z\ mod\ y))) \tag{v}$$

with the library lemmas (11) and

$$\forall x, y, z{:}\mathbb{N}\ \ z \neq 0 \wedge [x \equiv y]\ mod\ z \rightarrow z \mid (x - y) \tag{14}$$

$$\forall x, y, z, n{:}\mathbb{N}\ \ n \neq 0 \rightarrow [x + y \cdot (z\ mod\ n) \equiv x + y \cdot z\,]\ mod\ n\ . \tag{15}$$

We then command to use library lemma $\forall x, y, z{:}\mathbb{N}\ \ z \neq 0 \wedge x \leq y \rightarrow x \leq y \cdot z$ (with u substituted for x, x for y and $y - (z\ mod\ y)$ for z) after x factoring out, causing ✓eriFun to prove (v) with the synthesized lemma[5]

$$\forall x, y{:}\mathbb{N}\ \ y \neq 0 \rightarrow y > (x\ mod\ y)\ . \tag{16}$$

[4] At least one of $z \mid (x - y)$ or $z \mid (y - x)$ holds trivially because subtraction is defined such that $a - b = 0$ iff $a \leq b$.

[5] Synthesized lemmas are a spin-off of the system's termination analysis.

```
function ℑ_{N'}(x, k:ℕ):ℕ <=
if 2 > k
 then k
 else let h := ⌈k/2⌉; r := ℑ_{N'}((x mod 2 ↑ h), h); y := 2 ↑ k in
       (2 · r + ((r · r mod y) · x mod y) mod y)
      end_let
end_if

function ℑ_N(x, y:ℕ):ℕ <= if y ≠ 0 then y − ℑ_{N'}(x, log_2(y)) end_if
```

Fig. 5. Computation of multiplicative inverses by Newton-Raphson iteration

3.2 Newton's Method

Newton-Raphson iteration is a major tool in arbitrary-precision arithmetic and efficient algorithms for computing multiplicative inverses are developed in combination with Hensel Lifting [2]. Fig. 5 displays an implementation by procedure $\mathfrak{I}_N$ for odd numbers x and powers y of 2 (where $\uparrow$ computes exponentiation satisfying $0 \uparrow 0 = 1$). Procedure $\mathfrak{I}_N$ is defined via procedure $\mathfrak{I}_{N'}$ which is obtained from [3], viz. Algorithm 2′ *Recursive Hensel*, where however '$-$' instead of '$+$' is used in the result term. Algorithm 2′ was developed to compute a multiplicative inverse of x modulo p^k for any x not dividable by a prime p and returns a negative integer in most cases. By replacing '$-$' with '$+$', all calculations can be kept within $\mathbb{N}$ so that integer arithmetic is avoided. As procedure $\mathfrak{I}_{N'}$ computes the absolute value of a negative integer computed by Algorithm 2′, one additional subtraction is needed to obtain a multiplicative inverse which is implemented by procedure $\mathfrak{I}_N$. The computation of $\mathfrak{I}_N(x, 2^k)$ only requires *log* k steps (compared to k^2 steps for $\mathfrak{I}_B(x, 2^k)$), and therefore $\mathfrak{I}_N$ is the method of choice for computing a Montgomery Inverse.

However, Algorithm 2′ is flawed so that we wasted some time with our verification attempts: The four *mod*-calls in the algorithm are not needed for correctness, but care for efficiency as they keep the intermediate numbers small. Now instead of using modulus 2^k for both inner *mod*-calls, Algorithm 2′ calculates *mod* $2^{\lceil k/2 \rceil}$ thus spoiling correctness. As the flawed algorithm cares for even smaller numbers, the use of *mod* $2^{\lceil k/2 \rceil}$ could be beneficial indeed, and therefore it was not obvious to us whether we failed in the verification only because some mathematical argumentation was missing. But this consideration put us on the wrong track. Becoming eventually frustrated by the unsuccessful verification attempts, we started ✓eriFun's *Disprover* [1] which—to our surprise—came up with the counter example $x = 3, k = 2$ for Lemma 17 in less than a second.[6] We

[6] The Disprover is based on two heuristically controlled disproving calculi, and its implementation provides four selectable execution modes (Fast Search, Extended Search, Simple Terms and Structure Expansion). For difficult problems, the user may support the search for counter examples by presetting some of the universally quantified variables with general terms or concrete values.

then repaired the algorithm as displayed in Fig. 5 and subsequently verified it (cf. Lemma 20). Later we learned that the fault in Algorithm $2'$ has not been recognized so far and that one cannot do better to patch it as we did.[7]

For proving the inverse property (20) of procedure $\mathfrak{I}_N$, we first have to verify the correctness statement

$$\forall x, k{:}\mathbb{N}\ \ 2 \nmid x \to (x \cdot \mathfrak{I}_{N'}(x,k)\ mod\ 2^k) = 2^k - 1 \tag{17}$$

for procedure $\mathfrak{I}_{N'}$: We call the system to use induction corresponding to the recursion structure of procedure $\mathfrak{I}_{N'}$ which provides the induction hypothesis

$$\forall x'{:}\mathbb{N}\ \ k \geq 2 \wedge 2 \nmid x' \to (x' \cdot \mathfrak{I}_{N'}(x', \lceil k/2 \rceil)\ mod\ 2^{\lceil k/2 \rceil}) = 2^{\lceil k/2 \rceil} - 1\ . \tag{18}$$

✓`eriFun` proves the base case, but gets stuck in the step case with

$$\begin{array}{l} k \geq 2 \wedge 2 \nmid x \to \\ \quad (x \cdot (2A + (x \cdot (A^2\ mod\ 2^k)\ mod\ 2^k)\ mod\ 2^k)\ mod\ 2^k)\ \ = 2^k - 1 \end{array} \tag{i}$$

where A stands for $\mathfrak{I}_{N'}((x\ mod\ 2^{\lceil k/2 \rceil}), \lceil k/2 \rceil)$. By prompting the system to use Lemma 5, proof obligation (i) is simplified to

$$k \geq 2 \wedge 2 \nmid x \to (2B + B^2\ mod\ 2^k)\ \ = 2^k - 1 \tag{ii}$$

(where B abbreviates $x \cdot A$) thus eliminating the formal clutter resulting from the *mod*-calls in procedure $\mathfrak{I}_{N'}$. Next we replace $2B + B^2$ by $(B+1)^2 - 1$ and then call the system to replace B by $(B/C) \cdot C + R$ where $C = 2^{\lceil k/2 \rceil}$ and $R = ((x\ mod\ C) \cdot A\ mod\ C)$, which is justified by the quotient-remainder theorem as R rewrites to $(B\ mod\ C)$ by library lemma (5). This results in proof obligation

$$k \geq 2 \wedge 2 \nmid x \to (((B/C) \cdot C + R + 1)^2 - 1\ mod\ 2^k) = 2^k - 1 \tag{iii}$$

and we command to use the induction hypothesis (18) for replacing R in (iii) by $C - 1$. ✓`eriFun` then responds by computing

$$k \geq 2 \wedge 2 \nmid x \to (((B/C) \cdot C + C)^2 - 1\ mod\ 2^k) = 2^k - 1 \tag{iv}$$

using library lemmas $\forall x, y, z{:}\mathbb{N}\ y \neq 0 \wedge z \neq 0 \wedge z \mid y \to [(x\ mod\ y) \equiv x]\ mod\ z$ and (5) to prove $2 \nmid (x\ mod\ 2^{\lceil k/2 \rceil})$ for justifying the use of the induction hypothesis.

When instructed to factor out C in (iv), the system computes

$$k \geq 2 \wedge 2 \nmid x \to ((2^{\lceil k/2 \rceil})^2 \cdot (B/C + 1)^2 - 1\ mod\ 2^k) = 2^k - 1\ . \tag{v}$$

We command to use library lemma

$$\forall x, y, z{:}\mathbb{N}\ \ z \neq 0 \wedge z \nmid x \wedge z \mid y \wedge y \geq x \to (y - x\ mod\ z) = z - (x\ mod\ z) \tag{19}$$

for replacing the left-hand side of the equation in (v) yielding

$$k \geq 2 \wedge 2 \nmid x \to 2^k - (1\ mod\ 2^k) = 2^k - 1 \tag{vi}$$

[7] Personal communication with Jean-Guillaume Dumas.

justified by proof obligation

$$k \geq 2 \wedge 2 \nmid x \rightarrow \\ 2^k \neq 0 \wedge 2^k \nmid 1 \wedge 2^k \mid (2^{\lceil k/2 \rceil})^2 \cdot (B/C+1)^2 \wedge (2^{\lceil k/2 \rceil})^2 \cdot (B/C+1)^2 \geq 1$$

which ✓eriFun simplifies to

$$k \geq 2 \wedge 2 \nmid x \rightarrow 2^k \mid (2^{\lceil k/2 \rceil})^2 \cdot (B/C+1)^2 \tag{vii}$$

in a first step. It then uses auxiliary lemma $\forall x{:}\mathbb{N}\ x \leq 2 \cdot \lceil x/2 \rceil$ and the library lemmas (11) and $\forall x, y, z{:}\mathbb{N}\ x \neq 0 \wedge z \leq y \rightarrow x^z \mid x^y$ for rewriting (vii) subsequently to *true*. Finally the system simplifies (vi) to *true* as well by unfolding procedure *mod*, and Lemma 17 is proved.

When called to verify the inverse property

$$\forall x, y{:}\mathbb{N}\ \ 2 \nmid x \wedge 2^?(y) \rightarrow [x \cdot \mathfrak{I}_N(x,y) \equiv 1]\ mod\ y \tag{20}$$

of procedure $\mathfrak{I}_N$ (where $2^?(y)$ decides whether y is a power of 2), ✓eriFun unfolds the call of procedure $\mathfrak{I}_N$ and returns

$$y \geq 2 \wedge 2 \nmid x \wedge 2^?(y) \rightarrow (x \cdot y - x \cdot \mathfrak{I}_{N'}(x, log_2(y))\ mod\ y) = 1\ . \tag{viii}$$

Now we instruct the system to use library lemma (19) for replacing the left-hand side of the equation in (viii), and ✓eriFun computes

$$y \geq 2 \wedge 2 \nmid x \wedge 2^?(y) \rightarrow \\ (x \cdot \mathfrak{I}_{N'}(x, log_2(y))\ mod\ y) \neq 0 \wedge y - (x \cdot \mathfrak{I}_{N'}(x, log_2(y))\ mod\ y) = 1 \tag{ix}$$

using auxiliary lemma $\forall x, y{:}\mathbb{N}\ \ 2^?(y) \rightarrow y > \mathfrak{I}_{N'}(x, log_2(y))$ and the library lemmas (11), (14) and

$$\forall x, y, z{:}\mathbb{N}\ \ x \cdot y > x \cdot z \rightarrow y > z\ . \tag{21}$$

Finally we let the system use library lemma $\forall x{:}\mathbb{N}\ \ 2^?(x) \rightarrow 2^{log_2(x)} = x$ to replace both moduli y in (ix) by $2^{log_2(y)}$ causing ✓eriFun to rewrite both occurrences of $(x \cdot \mathfrak{I}_{N'}(x, log_2(y))\ mod\ y)$ with Lemma 17 to $y - 1$ and proof obligation (ix) to *true* in turn, thus completing the proof of (20).

4 Correctness of Montgomery Multiplication

We continue by defining procedures for computing the functions $\mathfrak{i}, h, \otimes$ and h^{-1} as displayed in Fig. 6, where we write $\mathfrak{i}(x, y)$ instead of $\mathfrak{i}_y(x)$ in the procedures and lemmas. As we aim to prove correctness of Montgomery Multiplication using procedure $\mathfrak{I}_N$ for computing the Montgomery Inverse with minimal costs, $2 \nmid n \wedge 2^?(m)$ instead of $gcd(n, m) = 1$ must be demanded to enable the use of Lemma 20 when proving the statements of Theorems 1 and 2. However, the multiplicative inverses n_m^{-1} and m_n^{-1} *both* are needed in the *proofs* (whereas

function i(x, y:$\mathbb{N}$):$\mathbb{N}$ <= if y $\neq$ 0 then (x $\cdot$ $^{-}$(y) mod y) end_if

function h(x, m, n:$\mathbb{N}$):$\mathbb{N}$ <= if n $\neq$ 0 then (x $\cdot$ m mod n) end_if

function $\otimes$(x, y, m, n:$\mathbb{N}$):$\mathbb{N}$ <= if n $\neq$ 0 then (x $\cdot$ y $\cdot$ $\mathfrak{I}$(m, n) mod n) end_if

function h^{-1}(x, m, n:$\mathbb{N}$):$\mathbb{N}$ <= if n $\neq$ 0 then (x $\cdot$ $\mathfrak{I}$(m, n) mod n) end_if

function $\mathfrak{I}$(x, y:$\mathbb{N}$):$\mathbb{N}$ <= if $2^?$(y) then $\mathfrak{I}_N$(x, y) else $\mathfrak{I}_B$(x, y) end_if

Fig. 6. Procedures for verifying Montgomery Multiplication

only n_m^{-1} is used in *applications* of redc and redc*). Consequently procedure $\mathfrak{I}_N$ cannot be used in the proofs as it obviously fails in computing m_n^{-1} (except for case $n = m = 1$, of course). This problem does not arise if procedure $\mathfrak{I}_B$ is used instead, where $gcd(n, m) = 1$ is demanded, because $\mathfrak{I}_B(n, m) = n_m^{-1}$ and $\mathfrak{I}_B(m, n) = m_n^{-1}$ for any coprimes n and m by Lemma 9. The replacement of $\mathfrak{I}_B$ by $\mathfrak{I}_N$ when computing the Montgomery Inverse then must be justified afterwards by additionally proving

$$\forall x, y{:}\mathbb{N}\ \ 2 \nmid x \wedge 2^?(y) \rightarrow \mathfrak{I}_B(x, y) = \mathfrak{I}_N(x, y)\ . \tag{22}$$

However, proving (22) would be a complicated and difficult enterprise because the recursion structures of procedures euclid and $\mathfrak{I}_{N'}$ differ significantly. But we can overcome this obstacle by a simple workaround: We use procedure $\mathfrak{I}$ of Fig. 6 instead of $\mathfrak{I}_B$ in the proofs and let the system verify the inverse property

$$\forall x, y{:}\mathbb{N}\ \ y \neq 0 \wedge gcd(x, y) = 1 \rightarrow [x \cdot \mathfrak{I}(x, y) \equiv 1]\ mod\ y \tag{1}$$

of procedure $\mathfrak{I}$ before: ✓eriFun easily succeeds with library lemma (4) and the inverse property (9) of procedure $\mathfrak{I}_B$ after being instructed to use library lemma $\forall x, y, n{:}\mathbb{N}\ \ n \geq 2 \wedge n \mid y \wedge gcd(x, y) = 1 \rightarrow n \nmid x$ and the inverse property (20) of procedure $\mathfrak{I}_N$. Consequently $\mathfrak{I}(n, m) = n_m^{-1}$ and $\mathfrak{I}(m, n) = m_n^{-1}$ for any coprimes n and m, and therefore $\mathfrak{I}$ can be used in the proofs. The use of $\mathfrak{I}_N$ instead of $\mathfrak{I}$ when computing the Montgomery Inverse is justified afterwards with lemma

$$\forall x, y : \mathbb{N}\ \ 2^?(y) \rightarrow \mathfrak{I}(x, y) = \mathfrak{I}_N(x, y)$$

having an obviously trivial (and automatic) proof.

Central for the proofs of Theorems 1 and 2 is the key property

$$\boxed{\begin{array}{l}\forall m, n, x{:}\mathbb{N}\quad m > n \wedge n \cdot m > x \wedge gcd(n, m) = 1 \rightarrow \\ \qquad\qquad redc(x, \mathsf{i}(\mathfrak{I}(n, m), m), m, n) = (x \cdot \mathfrak{I}(m, n)\ mod\ n)\end{array}} \tag{23}$$

of procedure redc: For proving Theorem 1.1

$$\begin{array}{l}\forall m, n, a{:}\mathbb{N}\ \ m > n > a \wedge gcd(n, m) = 1 \rightarrow \\ \quad h(a, m, n) = redc(a \cdot (m \cdot m\ mod\ n), \mathsf{i}(\mathfrak{I}(n, m), m), m, n)\end{array} \tag{Thm 1.1}$$

we command to use (23) for replacing the right-hand side of the equation by $a \cdot (m \cdot m \ mod \ n) \cdot \mathfrak{I}(m,n) \ mod \ n$. The system then replaces the left-hand side of the equation with $a \cdot m \ mod \ n$ by unfolding procedure call $h(a,m,n)$ and simplifies the resulting equation to *true* with Lemma 2, the synthesized lemma (16) and the library lemmas (5) and

$$\forall x,y,u,v{:}\mathbb{N} \ \ x > y \wedge u > v \rightarrow x \cdot u > y \cdot v \ . \tag{24}$$

Theorems 1.2 and 1.3, viz.

$$\begin{array}{l} \forall m,n,a,b{:}\mathbb{N} \ \ m > n > a \wedge n > b \wedge gcd(n,m) = 1 \\ \quad \rightarrow \otimes(a,b,m,n) = redc(a \cdot b, \mathsf{i}(\mathfrak{I}(n,m),m),m,n) \end{array} \tag{Thm 1.2}$$

$$\begin{array}{l} \forall m,n,a{:}\mathbb{N} \ \ m > n > a \wedge gcd(n,m) = 1 \\ \quad \rightarrow h^{-1}(a,m,n) = redc(a, \mathsf{i}(\mathfrak{I}(n,m),m),m,n) \end{array} \tag{Thm 1.3}$$

are (automatically) proved in the same way.

Having proved Theorem 1, it remains to verify the key property (23) for procedure redc (before we consider Theorem 2 subsequently). We start by proving that division by m in R_n can be expressed by $\mathfrak{I}$: We call the system to prove

$$\forall m,n,x{:}\mathbb{N} \ n \neq 0 \wedge m \mid x \wedge gcd(n,m) = 1 \rightarrow [x/m \equiv x \cdot \mathfrak{I}(m,n)] \ mod \ n \tag{25}$$

and $\checkmark$`eriFun` automatically succeeds with Lemma 2 and the library lemmas (4) and $\forall x,y,z{:}\mathbb{N} \ \ y \neq 0 \wedge y \mid x \rightarrow (x/y) \cdot y = x$.

As a consequence of Lemma 25, the quotient q in procedure redc can be expressed in R_n by $\mathfrak{I}$ in particular (if redc is called with the Montgomery Inverse as actual parameter for the formal parameter z), which is stated by lemma

$$\begin{array}{l} \forall m,n,x{:}\mathbb{N} \ \ n \neq 0 \wedge gcd(n,m) = 1 \\ \quad \rightarrow [(x + n \cdot (x \cdot \mathsf{i}(\mathfrak{I}(n,m),m) \ mod \ m))/m \equiv x \cdot \mathfrak{I}(m,n)] \ mod \ n \ . \end{array} \tag{26}$$

For obtaining a proof, we command to use Lemma 25 for replacing the left-hand side of the congruence in (26) by $(x + n \cdot (x \cdot \mathsf{i}(\mathfrak{I}(n,m),m) \ mod \ m)) \cdot \mathfrak{I}(m,n)$ causing $\checkmark$`eriFun` to complete the proof using Lemma 3 as well as the library lemmas (5), (10), (11), (15) and $\forall x,y{:}\mathbb{N} \ y \neq 0 \rightarrow y \mid (x + (y-1) \cdot x)$.

An obvious correctness demand for the method is that each call of redc (under the given requirements) computes some element of the residue class $mod \ n$. This is guaranteed by the conditional subtraction of n from the quotient q in the body of procedure redc. However, at most one subtraction of n from q results in the desired property only if $n + n > q$ holds, which is formulated by lemma

$$\forall m,n,x{:}\mathbb{N} \ \ m \cdot n > x \rightarrow n + n > (x + n \cdot (x \cdot \mathsf{i}(\mathfrak{I}(n,m),m) \ mod \ m)/m \ . \tag{27}$$

We prompt the system to use a case analysis upon $m \cdot (n+n) > x + n \cdot (x \cdot \mathsf{i}(\mathfrak{I}(n,m),m) \ mod \ m)$ causing $\checkmark$`eriFun` to prove the statement in the positive case with the library lemmas (5) and $\forall x,y,z{:}\mathbb{N} \ \ x \cdot z > y \rightarrow x > y/z$ and to

verify it in the negative case with the synthesized lemma (16) and the library lemmas (5), (21) and $\forall x, y, u, v{:}\mathbb{N}\ \ x > y \wedge u \geq v \rightarrow x + u > y + v$.

Now the *mod n* property of procedure redc can be verified by proving lemma

$$\begin{array}{l} \forall m, n, x{:}\mathbb{N}\ \ m > n \wedge n \cdot m > x \wedge gcd(n, m) = 1 \rightarrow \\ \quad redc(x, \mathfrak{i}(\mathfrak{I}(n, m), m), m, n) = (redc(x, \mathfrak{i}(\mathfrak{I}(n, m), m), m, n)\ mod\ n)\ . \end{array} \tag{28}$$

We let the system unfold the call of procedure *mod* in (28) causing ✓eriFun to use the synthesized lemma (16) for computing the simplified proof obligation

$$m > n \wedge n \cdot m > x \wedge gcd(n, m) = 1 \rightarrow n > redc(x, \mathfrak{i}(\mathfrak{I}(n, m), m), m, n)\ . \tag{i}$$

Then we command to unfold the call of procedure redc which simplifies to

$$\begin{array}{l} m > n \wedge n \cdot m > x \wedge gcd(n, m) = 1 \wedge \\ \quad (x + n \cdot (x \cdot \mathfrak{i}(\mathfrak{I}(n, m), m)\ mod\ m))/m \geq n \\ \qquad \rightarrow n > (x + n \cdot (x \cdot \mathfrak{i}(\mathfrak{I}(n, m), m)\ mod\ m))/m - n\ . \end{array} \tag{ii}$$

Finally we let the system use library lemma $\forall x, y, z{:}\mathbb{N}\ x > y \wedge y \geq z \rightarrow x - z > y - z$ resulting in proof obligation

$$\begin{array}{ll} & m > n \wedge n \cdot m > x \wedge gcd(n, m) = 1 \\ \wedge & (x + n \cdot (x \cdot \mathfrak{i}(\mathfrak{I}(n, m), m)\ mod\ m))/m \geq n \\ & [\ n + n > (x + n \cdot (x \cdot \mathfrak{i}(\mathfrak{I}(n, m), m)\ mod\ m))/m \wedge \\ \wedge & \quad (x + n \cdot (x \cdot \mathfrak{i}(\mathfrak{I}(n, m), m)\ mod\ m))/m \geq n \\ & \quad \rightarrow (n + n) - n > (x + n \cdot (x \cdot \mathfrak{i}(\mathfrak{I}(n, m), m)\ mod\ m))/m - n\ ] \\ \rightarrow & n > (x + n \cdot (x \cdot \mathfrak{i}(\mathfrak{I}(n, m), m)\ mod\ m))/m - n \end{array} \tag{iii}$$

which simplifies to

$$\begin{array}{ll} & m > n \wedge n \cdot m > x \wedge gcd(n, m) = 1 \\ \wedge & (x + n \cdot (x \cdot \mathfrak{i}(\mathfrak{I}(n, m), m)\ mod\ m))/m \geq n \\ \wedge & (n + n) - n > (x + n \cdot (x \cdot \mathfrak{i}(\mathfrak{I}(n, m), m)\ mod\ m))/m - n \\ \rightarrow & n > (x + n \cdot (x \cdot \mathfrak{i}(\mathfrak{I}(n, m), m)\ mod\ m))/m - n \end{array} \tag{iv}$$

by Lemma 27 and to *true* in turn using the plus-minus cancellation.

Now all lemmas for proving the key lemma (23) are available: We demand to use Lemma 28 for replacing the left-hand side of the equation in (23) by $(redc(x, \mathfrak{i}(\mathfrak{I}(n, m), m), m, n)\ mod\ n)$ and to apply lemma (26) for replacing the right-hand side by $((x + n \cdot (x \cdot \mathfrak{i}(\mathfrak{I}(n, m), m)\ mod\ m))/m\ mod\ n)$ resulting in the simplified proof obligation

$$\begin{array}{l} m > n \wedge n \cdot m > x \wedge gcd(n, m) = 1 \rightarrow \\ \quad [redc(x, \mathfrak{i}(\mathfrak{I}(n, m), m), m, n) \equiv (x + n \cdot (x \cdot \mathfrak{i}(\mathfrak{I}(n, m), m)\ mod\ m))/m]\ mod\ n\ . \end{array} \tag{v}$$

Then we unfold the call of procedure redc causing the system to prove (v) with library lemma (5).

Having proved the key lemma (23), the proof of Theorem 2

$$\begin{array}{l}\forall m,n,a,j{:}\mathbb{N}\ \ m>n>a\wedge gcd(n,m)=1\rightarrow\\ \quad (a^j\ mod\ n)=redc(redc^*(redc(a\cdot M,I,m,n),I,m,n,j),I,m,n)\end{array}\qquad \text{(Thm 2)}$$

(where $M=(m\cdot m)\ mod\ n$ and $I=\mathfrak{i}(\mathfrak{J}(n,m),m)$) is easily obtained by support of a further lemma, viz.

$$\begin{array}{l}\forall m,n,a,j{:}\mathbb{N}\ \ m>n>a\wedge gcd(n,m)=1\rightarrow\\ \quad (m\cdot a^j\ mod\ n)=redc^*(redc(a\cdot M,I,m,n),I,m,n,j)\ .\end{array}\qquad (29)$$

When called to use Peano induction upon j for proving (29), ✓eriFun proves the base case and rewrites the step case with the induction hypothesis to

$$\begin{array}{l}m>n>a\wedge gcd(n,m)=1\wedge j\neq 0\rightarrow\\ (m\cdot a^{j-1}\cdot a\ mod\ n)=redc(redc(a\cdot M,I,m,n)\cdot(m\cdot a^{j-1}\ mod\ n),I,m,n)\end{array}\qquad \text{(vi)}$$

Then we command to replace both calls of redc with the key lemma (23) causing ✓eriFun to succeed with the lemmas (2), (5), (16) and (24).

Finally the system proves (Thm 2) using lemmas (2), (5), (16), (29) and library lemma $\forall x,y,z{:}\mathbb{N}\ x\neq 0\wedge y>z\rightarrow x\cdot y>z$ after being prompted use (Thm 1.3) for replacing the right-hand side of the equation in (Thm 2).

5 Discussion and Conclusion

We presented machine assisted proofs verifying an efficient implementation of Montgomery Multiplication, where we developed the proofs ourselves as we are not aware of respective proofs published elsewhere. Our work also uncovered a serious fault in a published algorithm for computing multiplicative inverses based on Newton-Raphson Iteration [3], which could have dangerous consequences (particularly when used in cryptographic applications) if remained undetected.

Fig. 7 displays the effort for obtaining the proofs (including all procedures and lemmas which had been imported from our arithmetic proof library). Column *Proc.* counts the number of user defined procedures (the recursively defined ones given in parentheses), *Lem.* is the number of user defined lemmas (the number of synthesized lemmas given in parentheses), and *Rules* counts the total number of *HPL*-proof rule applications, separated into user invoked (*User*) and system initiated (*System*) ones (with the number of uses of *Induction* given in parentheses). Column *%* gives the automation degree, i.e. the ratio between *System* and *Rules*, *Steps* lists the number of first-order proof steps performed by the Symbolic Evaluator and *Time* displays the runtime of the Symbolic Evaluator.[8]

The first two rows show the effort for proving Lemmas 9 and 20 as illustrated in Sec. 3. As it can be observed from the numbers, verifying the computation of

[8] Time refers to running ✓eriFun 3.5 under *Windows 7 Enterprise* with an INTEL Core i7-2640M 2.80 GHz CPU using Java 1.8.0_45.

	Proc.	*Lem.*	*Rules*	*User*	*System*	*%*	*Steps*	*mm:ss*
$\mathfrak{I}_B(n,m) = n_m^{-1}$	8 (7)	49 (3)	241 (39)	36 (3)	205 (36)	85, 1 (92, 3)	3171	0:19
$\mathfrak{I}_N(n,m) = n_m^{-1}$	10 (9)	76 (3)	368 (59)	59 (3)	309 (56)	84, 0 (94, 9)	6692	1:32
Theorems 1 & 2	20 (12)	116 (3)	547 (78)	96 (6)	451 (72)	82, 4 (92, 3)	9739	2:19

Fig. 7. Proof statistics

multiplicative inverses by Newton-Raphson Iteration is much more challenging for the system and for the user than the method based on Bézout's Lemma. Row *Theorems 1 & 2* below displays the effort for proving Theorems 1 and 2 as illustrated in Sec. 4 (with the effort for the proofs of Lemmas 9 and 20 included).

The numbers in Fig. 7 almost coincide with the statistics obtained for other case studies in Number Theory performed with the system (see e.g. [14] and also [7] for more examples), viz. an automation degree of $\sim 85\%$ and a success rate of $\sim 95\%$ for the induction heuristic. All termination proofs (hence all required induction axioms in turn) had been obtained without user support, where 6 of the 12 recursively defined procedures, viz. mod, /, gcd, log_2, euclid and $\mathfrak{I}_{N'}$, do not terminate by structural recursion.[9] While an automation degree up to 100% can be achieved in mathematically simple domains, e.g. when sorting lists [7, 9], values of 85% and below are not that satisfying when concerned with *automated* reasoning. The cause is that quite often elaborate ideas for developing a proof are needed in Number Theory which are beyond the ability of the system's heuristics guiding the proof search.[10] We also are not aware of other reasoning systems offering more machine support for obtaining proofs in this difficult domain.

From the user's perspective, this case study necessitated more work than expected, and it was a novel experience for us to spend some effort for verifying a very small and non-recursively defined procedure. The reason is that correctness of procedure redc depends on some non-obvious and tricky number theoretic principles which made it difficult to spot the required lemmas. In fact, almost all effort was spend for the invention of the auxiliary lemmas in Sec. 4 and of Lemma 12 in Sec. 3.1. Once the "right" lemma for verifying a given proof obligation eventually was found, its proof turned out to be a routine task. The proof of Lemma 17 is an exception as it required some thoughts to create it and some effort as well to lead the system (thus spoiling the proof statistics). Proof development was significantly supported by the system's *Disprover* [1] which (besides detecting the fault in Algorithm $2'$) often helped not to waste time with trying to prove a false conjecture, where the computed counterexamples provided useful hints how to debug a lemma draft.

[9] Procedure $2^?(\ldots)$ is not user defined, but synthesized as the *domain procedure* [12] of the incompletely defined procedure log_2.

[10] Examples are the use of the quotient-remainder theorem for proving (i) in Sec. 3.1 and (iii) in Sec. 3.2 which are the essential proof steps there although more complex proof obligations result.

References

1. M. Aderhold, C. Walther, D. Szallies, and A. Schlosser. A Fast Disprover for ✓`veriFun`. In W. Ahrendt, P. Baumgartner, and H. de Nivelle, eds., *Proc. Workshop on Non-Theorems, Non-Validity, Non-Provability (DISPROVING-06)*, pages 59–69, Seattle, WA, 2006. http://verifun.de/documents.
2. R. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, New York, NY, USA, 2010.
3. J. Dumas. On Newton-Raphson Iteration for Multiplicative Inverses Modulo Prime Powers. *IEEE Trans. Computers*, 63(8):2106–2109, 2014. https://doi.org/10.1109/TC.2013.94.
4. D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus (NJ), USA, 2003.
5. A. J. Menezes, P. C. V. Oorschot, S. A. Vanstone, and R. L. Rivest. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton (FL), USA, 2001.
6. P. L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, 1985. https://doi.org/10.1090/S0025-5718-1985-0777282-X.
7. VeriFun. http://www.verifun.de.
8. C. Walther. On Proving the Termination of Algorithms by Machine. *Artificial Intelligence*, 71(1):101–157, 1994. https://doi.org/10.1016/0004-3702(94)90063-9.
9. C. Walther. A Largely Automated Verification of GHC's Natural Mergesort. Technical Report VFR 17/01, FB Informatik, Techn. Universität Darmstadt, 2017.
10. C. Walther and S. Schweitzer. Verification in the Classroom. *J. Autom. Reasoning*, 32(1):35–73, 2004. https://doi.org/10.1023/B:JARS.0000021872.64036.41.
11. C. Walther and S. Schweitzer. Automated Termination Analysis for Incompletely Defined Programs. In F. Baader and A. Voronkov, editors, *Proc. of the 11th Inter. Conf. on Logic for Progr., Artif. Intell. and Reasoning (LPAR-11)*, volume 3452 of *Lect. Notes in Artif. Intell.*, pages 332–346, Montevideo, Uruguay, 2005. Springer. https://doi.org/10.1007/978-3-540-32275-7_22.
12. C. Walther and S. Schweitzer. Reasoning about Incompletely Defined Programs. In G. Sutcliffe and A. Voronkov, editors, *Proc. of the 12th Inter. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-12)*, volume 3835 of *Lect. Notes in Artif. Intell.*, pages 427–442, Montego Bay, Jamaica, 2005. Springer. https://doi.org/10.1007/11591191_30.
13. C. Walther and S. Schweitzer. A Pragmatic Approach to Equality Reasoning. Technical Report VFR 06/02, FB Informatik, Technische Universität Darmstadt, 2006. http://verifun.de/documents.
14. C. Walther and N. Wasser. Fermat, Euler, Wilson - Three Case Studies in Number Theory. *J. Autom. Reasoning*, 59(2):267–286, 2017. https://doi.org/10.1007/s10817-016-9387-z.