A purely functional library for modular arithmetic and its application to certifying large prime numbers

Benjamin Grégoire and Laurent Théry

INRIA Sophia-Antipolis, France
[Benjamin.Gregoire|Laurent.Thery]@sophia.inria.fr

Abstract. Computing efficiently with numbers can be crucial for some theorem proving applications. In this paper, we present a library of modular arithmetic that has been developed within the Coq proof assistant. The library proposes the usual operations that have all been proved correct. The library is purely functional but can also be used on top of some native modular arithmetic. With this library, we have been capable of certifying the primality of numbers with more than 13000 digits.

1 Safe computation and theorem proving

Recent formalisations such as the four colour theorem [**?**] and the Flyspeck project [**?**] have shown all the benefits one can get from having a formal system where both proving and computing are possible. In the Coq proof assistant [**?**], computation is provided by the logic. A direct application of having computation inside the logic is the so-called two-level approach [**?**]. To illustrate it, let us consider the problem of proving the primality of some natural numbers. Suppose that we have defined a predicate *prime*: a number is prime if it has exactly two divisors, 1 and itself. How do we now prove that 17 is prime? The usual approach is to interactively build a proof object using tactics. Of course, this task can be automated by writing an ad-hoc tactic. Still, behind the scene, the system will have to build a proof object and the larger the number to be proved prime is, the larger the proof term will be. The two-level approach proposes an alternative strategy in two steps. In the first step, one writes a semi-decision procedure for the problem in the programming language of Coq. In our case, it amounts to writing a function *test* from natural numbers to booleans such that if the function returns *true* then the number is prime. For example, if the natural number is n, the function can check that there is no divisor between 2 and $\sqrt{n}$ by a simple iteration. In the second step, one proves that the function meets its specification. This means proving for our function *test* that

$$\forall n, \mathit{test}\ n = \texttt{true} \rightarrow \mathit{prime}\ n$$

Note that implication is sufficient for the two-level approach to work. Proving equivalence would not be of much interest here. A better way of proving that

a number n is not prime is to find externally a factor p of n and only check internally that p divides n.

Once the second step has been completed, for 17 to be certified as prime, it is sufficient to prove that the function *test* applied to 17 returns `true`. As the function *test* directly evaluates inside Coq, this last proof is simply the reflexivity of equality. Using the two-level approach, we have just transformed the problem of building a large proof object into a conversion problem: showing that *test* 17 is convertible to `true`. The size of the proof object is then independent of the number to be proved prime.

A recent improvement of the evaluation mechanism [**?**] has made the two-level approach much more attractive. The evaluation inside Coq is now as fast as the bytecode evaluation of the Ocaml language [**?**]. The only restriction when writing programs inside Coq is that programs must be purely functional, i.e. side effects are not allowed, and must always terminate. This is the price to pay to safely combine proofs and computations. Obviously, for this approach to be used, the Coq system should provide efficient functional implementations for the usual data structures: numbers, strings, vectors, hash tables, ...

The contribution of this paper is to propose a purely functional library to compute efficiently with large numbers inside Coq. The key idea of the library is to implement a representation of numbers that accommodates the divide and conquer strategy to speed up computation. The paper is organised as follows. In Section **??**, we present the current arithmetic of Coq and explain why a new representation of numbers is needed in order to compute efficiently with large numbers. In Section **??**, we give an overview of our new library based on this new representation. In Section **??**, we detail two possible instantiations of the library. Finally, Section **??** presents an application of the library to the particular problem of certifying large prime numbers.

2 Linear versus tree representation of numbers

In the standard library of Coq, strictly positive numbers are represented as linear structures, low bits first.

```
Inductive positive : Set :=
  | xI : positive -> positive
  | xO : positive -> positive
  | xH : positive.
```

`xH` is 1, `(xO p)` is two times the value of `p` and `(xI p)` is two times plus one the value of `p`. For example, 17 and 18 are represented as `xI (xO (xO (xO (xH))))` and `xO (xI (xO (xO (xH))))` respectively. The choice of the representation has some direct impact on the way operations are implemented. To illustrate this on an example, let us consider the comparison function `Pcmp`. It takes two positive numbers and returns a comparison value

```
Inductive comparison: Set := Eq | Lt | Gt.
```

As numbers are represented low bits first, to compare two numbers one needs to walk down both numbers keeping track of what the current status of the comparison is. This is what the auxiliary function `Pcompare` does. The main function `Pcomp` starts the computation with the initial status being equality.

```
Fixpoint Pcompare (x y: positive) (r: comparison): comparison :=
  match x, y with
  |    xH,    xH => r
  |    xH,    _  => Lt
  |    _ ,    xH => Gt
  | xI x', xI y' => Pcompare x' y' r
  | xO x', xO y' => Pcompare x' y' r
  | xI x', xO y' => Pcompare x' y' Gt
  | xO x', xI y' => Pcompare x' y' Lt
  end.

Definition Pcmp x y := Pcompare x y Eq.
```

This is clearly not optimal but is the best one can do with this representation: recursive calls only skip a single bit. Efficient algorithms for large numbers, like Karatsuba multiplication [**?**], use a divide and conquer strategy. They require to be able to split numbers in parts efficiently. This motivates our representation based on a tree-like structure. Given an arbitrary one-word set `w`, we define the two-word set `w2 w` as follows

```
Inductive w2 (w: Set): Set :=  WW : w -> w -> w2 w.
```

For example, `(WW true false)` is of type `(w2 bool)`. We choose in an arbitrary way that high bits are the first argument of `WW`, low bits the second one. Now we use a recursive type definition and define the type of numbers of height `n` as

```
Fixpoint word (w: Set) (n:nat): Set :=
 match n with
 |   O => w
 | S n => w2 (word w n)
 end.
```

An object of type `(word w n)` is a complete binary tree that contains $2^{\mathtt{n}}$ objects of type `w`. Given a number, one has to choose an appropriate height to represent it exactly. For example, taking the usual booleans for base words, a minimum height of 2 is necessary to represent the number 13. With this height, numbers have type `(word bool 2)` and `(WW (WW true true) (WW false true))` denotes the number 13.

Arithmetic operations are not going to be defined on the type `word` directly. We use a technique similar to the one in [**?**]. A functor is first defined that allows to build a two-word modular arithmetic on top of a single-word one. The functor is then applied iteratively to get the final implementation. In the following, x, y are used to denote one-word variables and xx, yy to denote two-word variables.

When defining a new function f, we just need to explain how to compute the result on two-word values knowing how to compute it on one-word values. We use the notation w_f for the single-word version of f and ww_f for the two-word version. For example, let us go back to our comparison function `Pcompare` and try to define it on our trees. We first suppose the existence of the comparison on single words

```
Variable w_compare: w -> w -> comparison -> comparison.
```

and then define the function for two-word values

```
Definition ww_compare (xx yy: w2 w) (r: comparison) :=
  match xx, yy with
    WW xH xL, WW yH yL => w_compare xH yH (w_compare xL yL r)
  end.
```

This is not the function that is in our library. Instead, we can take advantage of the tree-like structure and compare high bits first.

```
Variable w_cmp: w -> w -> comparison.

Definition ww_cmp (xx yy: w2 w) :=
  match xx, yy with
    WW xH xL, WW yH yL =>
     match w_cmp xH yH with Eq => w_cmp xL yL | cmp => cmp end
  end.
```

The key property of our representation is that splitting number in two is for free. The next section details why this property is crucial to implement efficient algorithms for functions like multiplication, division and square root. Note that, in term of memory allocation, having a tree structure does not produce any overhead. In a functional setting, building a binary tree structure or building the equivalent linear list of words requires the same number of cells.

One main drawback of our representation is that we manipulate only complete binary trees. So, even if we choose carefully the appropriate height, half of the words could be unnecessary to compute the final result. To soften this problem, we have extended the definition of `w2` to include an empty word `W0`.

```
Inductive w2 (w: Set): Set :=
|  W0: w2
|  WW: w -> w -> w2.
```

For example, the number 13 can be represented at height 3 as

```
 WW W0 (WW (WW true true) (WW false true))
```

With this extension, we lose uniqueness of representation. Still, there is a notion of canonicity, `W0` should always be preferred to a sub-tree full of zeros. Note that, in our development, all functions have been carefully written in order to

preserve canonicity, but canonicity is not part of their specification since it is not necessary to ensure safe computations. Using `w_0` to represent the one-word zero, the final version of the comparison function is then

```
Definition ww_cmp (xx yy: w2 w) :=
  match xx, yy with
  | W0, W0 => Eq
  | W0, WW yH yL =>
     match w_cmp w_0 yH  with Eq => w_cmp w_0 yL  | _   => Lt  end
  | WW xh xl, W0 =>
     match w_cmp xH  w_0 with Eq => w_cmp xL  w_0 | _   => Gt  end
  | WW xH xL, WW yH yL =>
     match w_cmp xH  yH  with Eq => w_cmp xL  yL  | cmp => cmp end
  end.
```

3 The certified library

Our library includes the usual functions: comparison, successor, predecessor, opposite, addition, subtraction, multiplication, square, Euclidean division, modulo, integer square root, gcd, and power. It is a modular library: we manipulate trees (or words) of the same height (resp. of the same size). For addition and subtraction, we also provide an exact version that returns a word and a carry. For multiplication, we also provide an exact version returning two words.

Since we want to use our library in the context of the two-level approach, we must carefully choose the algorithms we implement. Furthermore, semi-decision procedures must also be certified, so every function of our library must come along with its proof of correctness.

Specifications are expressed using predicates over integers. For this, we use two interpretation functions [| |] and [[ ]]. Given a one-word element x, its corresponding integer value is $[|x|]$. Given a two-word element xx, its corresponding integer value is $[[xx]]$. The base of the arithmetic, i.e. one plus the maximum value that fits in a single-word, is `wB`. We write `w_0` (resp. `w_1`) for the word with corresponding integer value 0 (resp. 1). From these definitions, the following statement holds

$$\forall x\, y,\ [[\texttt{WW}\ x\ y\,]] = [|x|] * \texttt{wB} + [|y|]$$

Once a function is defined, its correctness has to be proved. For example, for the comparison defined in the previous section, one needs to prove that if the function `w_cmp` meets its specification

$$\begin{array}{l}\forall x\, y,\ \mathit{match}\ \texttt{w\_cmp}\ x\ y\ \mathit{with}\\ \quad \mid \texttt{Eq}\ \rightarrow [|x|] = [|y|] \mid \texttt{Lt}\ \rightarrow [|x|] < [|y|] \mid \texttt{Gt}\ \rightarrow [|x|] > [|y|]\\ \quad \mathit{end}\end{array}$$

so does the function `ww_cmp`

$$\begin{array}{l}\forall xx\, yy,\ \mathit{match}\ \texttt{ww\_cmp}\ xx\ yy\ \mathit{with}\\ \quad \mid \texttt{Eq}\ \rightarrow [[xx]] = [[yy]] \mid \texttt{Lt}\ \rightarrow [[xx]] < [[yy]] \mid \texttt{Gt}\ \rightarrow [[xx]] > [[yy]]\\ \quad \mathit{end}\end{array}$$

3.1 Words and carries

Carries are important for operations like addition and subtraction. In our functional setting, carries encapsulate words

```
Inductive carry (w: Set): Set :=
  | C0: w -> carry
  | C1: w -> carry.
```

Two interpretation functions [+| |] and [-| |] are associated with carries. One interprets the carry positively: [+|C1 x|] = wB + [|x|] and [+|C0 x|] = [|x|]. The other interprets it negatively (i.e. a borrow): [-|C1 x|] = [|x|] − wB and [-|C0 x|] = [|x|]. To illustrate how carries are manipulated, let us consider the successor function. In our library, it is represented by two functions

```
w_succ: w -> w
w_succ_c: w -> carry w
```

The first function represents the modular version, the second the exact version. With these two functions, it is possible to define the version for two-word elements. For example, the definition for the modular version is

```
Definition ww_succ xx :=
 match xx with
 | W0 => WW w_0 w_1
 | WW xH xL =>
   match w_succ_c xL with
   | C0 l => WW xH l
   | C1 l => WW (w_succ xH) w_0
   end
 end.
```

Note that, unlike what happens in imperative languages, returning a carry allocates a memory cell. So in our implementation we avoid as much as possible to create them. When we know in advance that the result always returns (resp. does not return) a carry, we can call the modular function instead. An example of such a situation is a naive implementation of the exact function that adds 2 to a one-word element by calling twice the successor function:

```
Definition w_add2 x :=
 match w_succ_c x with
 | C0 y => w_succ_c y
 | C1 y => C1 (w_succ y)
 end.
```

In the case when the first increment has created a carry, we are sure that the second increment cannot raise any carry, so we can directly call the function `w_succ`. Also, we use a combination of partial evaluation and continuation passing style to get shorter definitions. This has proved to ease considerably the proving phase without changing the efficiency of functions.

3.2 Shifting bits

If most of the operations work at word level, some functions (like the shifting operation) require to work at a lower level, i.e. the bit level. Surprisingly, all the operations we had to perform at bit level can be built using a single function

```
w_add_mul_div : positive -> w -> w -> w
```

Evaluating `(w_add_mul_div p x y)` returns a new word that is composed for its last `p` bits by the first bits of `y` and for the remaining bits by the last bits of `x`. Its specification is

$$\forall p\, x\, y,\ 2^p < \mathtt{wB} \Rightarrow \\ [|\mathtt{w\_add\_mul\_div}\, p\, x\, y|] = ([|x|] * 2^p + ([|y|] * 2^p)/\mathtt{wB}) \bmod \mathtt{wB}$$

Two degenerated versions of this function are of direct interest. Calling it with `w_0` as second argument implements the shift left. Calling it with `w_0` as first argument implements the shift right.

3.3 Divide and conquer algorithms

Karatsuba multiplication Speeding up the multiplication was the main motivation of our tree representation for numbers. The multiplication is represented in our library by the function

```
w_mul_c: w -> w -> w2 w
```

and its specification is

$$\forall x\, y,\ [[\mathtt{w\_mul\_c}\, x\, y]] = [|x|] * [|y|]$$

The naive implementation on two-word elements follows the simple equation

$$[[\mathtt{WW}\ x_h\ x_l\ ]] * [[\mathtt{WW}\ y_h\ y_l\ ]] = \\ [|x_h|] * [|y_h|] * \mathtt{wB}^2 + ([|x_h|] * [|y_l|] + [|x_l|] * [|y_h|]) * \mathtt{wB} + [|x_l|] * [|y_l|]$$

Thus, performing a multiplication requires four submultiplications. Karatsuba multiplication [**?**] saves one of these submultiplications

$$[[\mathtt{WW}\ x_h\ x_l\ ]] * [[\mathtt{WW}\ y_h\ y_l\ ]] = \\ \quad \mathit{let}\ h\ = [|x_h|] * [|y_h|]\ \mathit{in} \\ \quad \mathit{let}\ l\ = [|x_l|] * [|y_l|]\ \mathit{in} \\ \quad h * \mathtt{wB}^2 + ((h + l) - ([|x_h|] - [|x_l|]) * ([|y_h|] - [|y_l|])) * \mathtt{wB} + l$$

Karatsuba multiplication is more efficient than the naive one only when numbers are large enough. So our library includes both implementations for multiplication. They are used separately to define two different functors. The functor with the naive multiplication is only used for trees of "small" height.

Recursive Division The general Euclidean division algorithm that we have used is the usual schoolboy method that iterates the division of two words by one word. It is then crucial to perform this two-by-one division efficiently. The algorithm we have implemented is the one presented in [**?**]. The idea is to use the recursive call on high bits to guess an approximation of the quotient and then to perform an appropriate adjustment to get the exact quotient.

In our development, the two-by-one division takes three words and returns a pair composed of the quotient and the remainder

```
Variable w_div21: w -> w -> w -> w * w
```

and its specification is

$$\forall x_1\ x_2\ y,\ let\ q,\ r = \texttt{w\_div21}\ x_1\ x_2\ y\ in$$
$$[|x_1|] < [|y|] \Rightarrow \texttt{wB}/2 \leq [|y|] \Rightarrow [[\texttt{WW}\ x_1\ x_2\,]] = [|q|] * [|y|] + [|r|] \wedge 0 \leq [|r|] < [|y|]$$

The two conditions deserve some explanation. The first one ensures that the quotient fits in one word. The second one ensures that the recursive call computes an approximation of the quotient that is not too far from the correct value.

Before defining the function `ww_div21` for two-word elements, we need to define the intermediate function `w_div32` that divides three one-word elements by two one-word elements. Its specification is

$$\begin{array}{l}\forall x_1\ x_2\ x_3\ y_1\ y_2,\ let\ q,\ rr = \texttt{w\_div32}\ x_1\ x_2\ x_3\ y_1\ y_2\ in \\ \quad [[\texttt{WW}\ x_1\ x_2\,]] < [[\texttt{WW}\ y_1\ y_2\,]] \Rightarrow \texttt{wB}/2 \leq [|y_1|] \Rightarrow \\ \quad\quad [|x_1|] * \texttt{wB}^2 + [|x_2|] * \texttt{wB} + [|x_3|] = [|q|] * [[\texttt{WW}\ y_1\ y_2\,]] + [[rr]] \wedge \\ \quad\quad 0 \leq [[rr]] < [[\texttt{WW}\ y_1\ y_2\,]]\end{array}$$

The two conditions play the same roles as the ones in the specification of `w_div21`. As the code is a bit intricate, here we just explain how the function proceeds. It first calls `w_div21` to divide x_1 and x_2 by y_1. This gives a pair (q, r) such that

$$[|x_1|] * \texttt{wB} + [|x_2|] = [|q|] * [|y_1|] + [|r|]$$

q is considered as the approximation of the final quotient. The condition $\texttt{wB}/2 \leq [|y_1|]$ ensures that if this approximation is not exact, then it exceeds the real value of at most two units. So the quotient can only be q, $q-1$ or $q-2$. As we have

$$[|x_1|] * \texttt{wB}^2 + [|x_2|] * \texttt{wB} + [|x_3|] = [|q|] * [[\texttt{WW}\ y_1\ y_2\,]] + ([[\texttt{WW}\ r\ x_3\,]] - [|q|] * [|y_2|])$$

we know in which situation we are by testing the sign of the candidate remainder. In our modular arithmetic, it amounts to checking whether or not the subtraction of (`w_mul_c` $q\ y_2$) from (`WW` $r\ x_3$) produces a borrow. If it is positive or zero (no borrow), the quotient is q. If it is negative (a borrow), we have to consider $q-1$ and add in consequence (`WW` $y_1\ y_2$) to the candidate remainder. We test again the sign of this new candidate. If it is positive, the quotient is $q-1$, otherwise is $q-2$. The definition of `ww_div21` is now straightforward. Forgetting the `W0` constructor, we have

```
Definition ww_div21 xx1 xx2 yy :=
 match xx1, xx2, yy with
 ....
 | WW x1H x1L, WW x2H x2L, WW yH yL =>
       let (qH, rr) := w_div32 x1H x1L x2H yH yL in
       match rr with
       | W0 => (WW qH w_0, WW w_0 x2L)
       | WW rH rL =>
         let (qL, s) := w_div32 rH rL x2L yH yL in
         (WW qH qL, s)
       end
 end.
```

These two divisions can only be used if the divisor y is greater than equal to $\mathtt{wB}/2$. This is not restrictive because, if y is too small, we can always find an n such that $y * 2^n \geq \mathtt{wB}/2$. If we have $x * 2^n = q * (y * 2^n) + r$ for some x and r, then r can be written as $r = 2^n * r'$, so $x = q * y + r'$. Hence, to perform the division of two numbers of the same size, we first shift divisor and dividend by n. The shifted dividend fits in two words and its high part is smaller than the shifted divisor. Then, we use the two-by-one division. The resulting quotient is correct and we just have to unshift the remainder.

Recursive Square Root The algorithm for computing the integer square root is similar to the one for division. It was first described in [**?**] and has already been formalised in a theorem prover [**?**]. It requires the number to be split in four. For this reason it is represented by the following function in our library

```
w_sqrt2: w -> w -> w * carry w;
```

The function returns the integer square root and the rest. Its specification is

$$\begin{array}{l}\forall x\, y,\ \mathit{let}\ s, r = \mathtt{w\_sqrt2}\ x\ y\ \mathit{in}\\ \quad \mathtt{wB}/4 \leq [|x|] \Rightarrow [[\mathtt{WW}\ x\ y\,]] = [|s|]^2 + [^+|r|] \wedge [^+|r|] \leq 2 * [|s|]\end{array}$$

As for division, the input must be large enough so that the recursive call that computes the approximation is not too far from the exact value.

The definition of the square root needs a support function that implements a division by twice a number

```
w_div2s: carry w -> w -> w -> carry w * carry w
```

with its specification

$$\begin{array}{l}\forall x_1\ x_2\ y,\ \mathit{let}\ q, r = \mathtt{w\_div2s}\ x_1\ x_2\ y\ \mathit{in}\\ \quad \mathtt{wB}/2 \leq [|y|] \Rightarrow [^+|x_1|] \leq 2 * [|y|] \Rightarrow\\ \quad\quad [^+|x_1|] * \mathtt{wB} + [|x_2|] = [^+|q|] * (2 * [|y|]) + [^+|r|]\ \wedge\ 0 \leq [^+|r|] < 2 * [|y|]\end{array}$$

The idea of the algorithm is summarised by the following equation

$$\begin{array}{l} \mathit{let}\ q_h, r = \texttt{w\_sqrt2}\ x_h\ x_l \quad \mathit{in} \\ \mathit{let}\ q_l, r_1 = \texttt{w\_div2s}\ r\ y_h\ q_h\ \mathit{in} \\ [[\texttt{WW}\ x_h\ x_l\,]] * \texttt{wB}^2 + [[\texttt{WW}\ y_h\ y_l\,]] = [[\texttt{WW}\ q_h\ q_l\,]]^2 + ([+|r_1|] * \texttt{wB} + [|y_l|] - [|q_l|]^2) \end{array}$$

$(\texttt{WW}\ q_h\ q_l)$ is a candidate for the square root of $(\texttt{WW}\ (\texttt{WW}\ x_h\ x_l)\ (\texttt{WW}\ y_h\ y_l))$. Because of the condition on the input, we are sure that the integer square root is either $(\texttt{WW}\ q_h\ q_l)$ or $(\texttt{WW}\ q_h\ q_l) - 1$. It is the sign of $[+|r_1|] * \texttt{wB} + [|y_l|] - [|q_l|]^2$ that indicates which one to choose.

4 Implementing base word arithmetic

The final step to complete our library is to define the arithmetic for the base words. Once defined, we get the modular arithmetic for the desired size by applying an appropriate number of times our functors on top of this base arithmetic. In a classical implementation, these base words would be machine words. Unfortunately, machine words are not yet accessible from the Coq language.

4.1 Defined modular arithmetic

For the moment, the only way to have a modular arithmetic for base words inside Coq is to define base words as a datatype. For example, we have for two-bit words

```
Inductive word2 : Set := OO | OI | IO  | II.
```

The functions are then defined by simple case analysis. For example, the exact successor function is defined as

```
Definition word2_succ_c x :=
 match x with
 | OO => C0 OI
 | OI => C0 IO
 | IO => C0 II
 | II => C1 OO
 end.
```

We also need to give the proofs that every function meets its specification. These proofs are also done by case analysis.

Rather than writing by hand functions and proofs, we have written an Ocaml program instead. This program takes the word size as argument and generates the desired base arithmetic with all its proofs. It is a nice application of meta-proving. Unfortunately, functions and their corresponding proofs grow quickly with the word size. For example, the addition for `word8` is a pattern matching of 65536 cases. `word8` is actually the largest size Coq can handle.

The main benefit of this approach is to get an arithmetic library that is entirely expressed in the logic of Coq. The library is portable: no extension of the Coq kernel is needed.

4.2 Native modular arithmetic

To test our library with some machine word arithmetic, we use the extraction mechanism that converts automatically Coq functions into Ocaml functions. It is then possible to run the resulting program with the 31-bit native Ocaml arithmetic or a simulated 64-bit arithmetic. Not all the functions that we have implemented have their corresponding functions in the native modular arithmetic, so some native code had to be developed for these functions. The formal verification of this code is also possible and we did it for some of these functions. Running the extracted library with machine word arithmetic gives an idea of the speed-up we could get if we had a native arithmetic in Coq.

5 Evaluating the library

A way of applying the two-level approach for proving primality has been presented in [**?**]. It is based on the notion of prime certificate and more precisely of *Pocklington certificate*. A prime certificate is an object that witnesses the primality of a number. The Pocklington certificates we have been using are justified by the following theorem given in [**?**]:

Theorem 1. *Given a number n, a witness a and some pairs of natural numbers $(p_1, \alpha_1), \ldots, (p_k, \alpha_k)$ where all the p_i are prime numbers, let*

$$F_1 = p_1^{\alpha_1} \ldots p_k^{\alpha_k}$$
$$R_1 = (n-1)/F_1$$
$$s = R_1/(2F_1)$$
$$r = R_1 mod\ (2F_1)$$

it is sufficient for n to be prime that the following conditions hold:

$$F_1 \text{ is even, } R_1 \text{ is odd, and } F_1 R_1 = n - 1 \tag{1}$$
$$(F_1 + 1)(2F_1^2 + (r-1)F_1 + 1) > n \tag{2}$$
$$a^{n-1} = 1 (mod\ n) \tag{3}$$
$$\forall i \in \{1, \ldots, k\}\ \ \gcd(a^{\frac{n-1}{p_i}} - 1, n) = 1 \tag{4}$$
$$r^2 - 8s \text{ is not a square or } s = 0 \tag{5}$$

For a prime number n, the list $[a, p_1, \alpha_1, p_2, \alpha_2, \ldots, p_k, \alpha_k]$ represents its Pocklington certificate. Even if generating a certificate for a given n can be cpu-intensive, verifying conditions (1)-(5) is an order of magnitude simpler than evaluating $(test\ n)$ (computing $a^{n-1}(mod\ n)$ requires a maximum of $2log_2\ n$ modular multiplications). In fact, only the verification of conditions (1)-(5) is crucial for asserting primality. This requires safe computation and is done inside Coq. The generation of the certificate is delegated to an external tool. This is a direct application of the skeptic approach described in [**?**,**?**]. Note that this method of certifying prime numbers is effective only if the prime number n is such that $n - 1$ can be easily partially factorised.

With respect to the usual approach for the same problem [**?**], the two-level approach gives a significant improvement in terms of size of the proof object and in terms of time. Figure **??** illustrates this on some examples (P_{150} is a random prime number with 150 digits and the millennium prime is a prime number with 2000 digits discovered by John B. Cosgrave). However, due to the limitations of the linear representation of numbers in Coq, even with the two-level approach, we were not capable of certifying large prime numbers ($>$ 1000 digits) as illustrated by the millennium prime. The same occurred when applying the Lucas-Lehmer test for proving the primality of Mersenne numbers, i.e. numbers that can be written as $2^p - 1$.

Theorem 2. *Let (S_n) be recursively defined by $S_0 = 4$ and $S_{n+1} = S_n^2 - 2$. For $p > 2$, $2^p - 1$ is prime if and only if $(2^p - 1)|S_{p-2}$.*

The largest Mersenne number we could certify was $2^{4423} - 1$ that has 1332 digits.

The idea is then to use our new library based on a tree-like representation of numbers. The complete library with the corresponding contribution for prime numbers is available at `http://gforge.inria.fr/projects/coqprime/`. It consists of 9000 lines of hand-written definitions and proofs. The automatically generated `word8` arithmetic is much bigger, 95 Mb: 41 Mb are used to define functions and 54 Mb for the proofs. This is the largest ever contribution that has been verified by Coq. With this new library, we have been capable of proving that the Mersenne number $2^{44497} - 1$ was prime using Coq with the Lucas-Lehmer test. As far as we know, it is the largest prime number that has been certified by a theorem prover.

The certification with our library is faster even for small numbers. This is illustrated in Figure **??** and the fifth and sixth columns of Figure **??**. There is a maximum speed-up of 70. These benchmarks have been run on a Pentium 4 with 1 Gigabyte of RAM.

Comparing `word8` with the 31-bit Ocaml integer `w31` shows all the benefit we could get from having machine words in Coq. There is a maximum speed-up of 95 with respect to `word8`. This means a speed-up of 6650 with respect to the standard Coq library.

The 64-bit Ocaml integer `w64` is a simulated arithmetic (our processor has only 32 bits). This is why there is not such a gap between `w31` and `w64`. Big_int [**?**] is the standard exact library for Ocaml. It has a purely functional interface but is written in C. The comparison is not bad. For the last Mersenne, `w64` is only 4.6 times slower than Big_int. It is also very interesting that this gap is getting smaller as numbers get larger. On individual functions, random tests on addition give a ratio of 4 and on multiplication a ratio of 10.

We are still far away from getting the performance of the gmp [**?**] library. This library is written in C and uses in-place computation instead. This minimises considerably the number of memory allocations. Unfortunately, in-place computation is not compatible with the logic of Coq.

		size		time	
prime	digits	standard	two-level	standard	two-level
1234567891	10	94K	0.453K	3.98s	0.50s
74747474747474747	17	145K	0.502K	9.87s	0.56s
1111111111111111111	19	223K	0.664K	17.41s	0.66s
$(2^{148}+1)/17$	44	1.2M	0.798K	350.63s	2.77s
P_{150}	150	_	1.902K	_	75.62s
millennium prime	2000	_	_	_	_

Fig. 1. Some verifications of certificates with the standard and two-level approaches

	digits	positive	word8
1234567891	10	0.50s	0.10s
74747474747474747	17	0.56s	0.12s
1111111111111111111	19	0.66s	0.20s
$(2^{148}+1)/17$	44	2.77s	0.36s
P_{150}	150	75.62s	8.44s
millennium prime	2000	_	5320.05s

Fig. 2. Some verifications of certificates with the standard and our Coq arithmetics

#	n	digits	year	positive	word8	w31	w64	Big_int
12	127	39	1876	0.73s	0.04s	0.01s	0.s	0.s
13	521	157	1952	53.00s	1.85s	0.02s	0.02s	0.s
14	607	183	1952	84.00s	2.78s	0.03s	0.03s	0.s
15	1279	386	1952	827.00s	20.21s	0.25s	0.16s	0.02s
16	2203	664	1952	4421.00s	89.1s	1.1s	0.8s	0.08s
17	2281	687	1952	4964.00s	97.59s	1.21s	0.82s	0.09s
18	3217	969	1957	14680.00s	237.65s	2.85s	2.14s	0.22s
19	4253	1281	1961	35198.00s	494.09s	6.4s	4.58s	0.6s
20	4423	1332	1961	39766.00s	563.27s	6.99s	4.99s	0.67s
21	9689	2917	1963		5304.08s	56.1s	39.98s	5.89s
22	9941	2993	1963		5650.63s	60.5s	42.53s	6.32s
23	11213	3376	1963		7607.00s	80.56s	57.47s	11.25s
24	19937	6002	1971		34653.12s	377.24s	268.09s	45.75s
25	21701	6533	1978		43746.21s	463.02s	338.04s	58.56s
26	23209	6987	1979		51210.56s	538.33s	403.48s	88.43s
27	44497	13395	1979		282784.09s	3282.23s	2208.45s	476.75s

Fig. 3. Times to verify Mersenne numbers

6 Conclusions

The main contribution of our work is to present a certified library for performing modular arithmetic. Individual arithmetic functions have already been proved correct, see for example [**?**]. To our knowledge, it is the first time verification has been applied successfully to a complete library with non-trivial algorithms. Our motivation was to improve integer arithmetic inside Coq. The figures given in Section **??** show that this goal has been reached: we are now capable of manipulating numbers with more than 13000 digits. These tests also show all the benefit we could get from a native base arithmetic. We hope this will motivate researchers to integrate machine word arithmetic inside Coq.

Expressing the arithmetic in the logic has a price: no side effect is possible, also numbers are allocated progressively, not in one block. A natural continuation of our work would be to prove the correctness of a library with side effects. This would require a much more intensive verification work since in-place computing is known to be much harder to verify. Note that directly integrating an existing library inside the prover with no verification would go against the philosophy of Coq to keep its trusted computing base as small as possible.

From the methodological point of view, the most interesting aspect of this work has been the use of the meta-proving technique to generate our base arithmetic. This has proved to be a very powerful technique. Files for the base arithmetic are generated in an ad-hoc manner by concatenating strings. Developing a more adequate support for meta-proving inside the prover seems a very promising future work. Note that meta-proving could also be a solution to get more flexibility in the proof system. Slightly changing our representation, for example adding not only `W0` but also `W1` and `W-1` to the `w2` type, would break most of our definitions and proofs. Meta-proving could be a solution for having a formal development for a family of data structures rather than just a single one.

Finally, on December 2005, a new prime Mersenne number has been discovered: $2^{30402457} - 1$. It took five days to perform its Lucas-Lehmer test on a super computer. The program uses a very intriguing algorithm for multiplication [**?**]. Proving the correctness of such an algorithm seems a very challenging task.

Acknowledgments

We would like to thank the anonymous referees for their careful reading of the paper and specially the referee who suggested a simplification to our implementation of Karatsuba multiplication.