A computational approach to Pocklington certificates in type theory

Benjamin Grégoire[1], Laurent Théry[1], and Benjamin Werner[2]

[1] INRIA Sophia-Antipolis, France
`[Benjamin.Gregoire|Laurent.Thery]@sophia.inria.fr`
[2] INRIA Futurs, France `Benjamin.Werner@inria.fr`

Abstract. Pocklington certificates are known to provide short proofs of primality. We show how to perform this in the framework of formal, mechanically checked, proofs. We present an encoding of certificates for the proof system Coq which yields radically improved performances by relying heavily on computations inside and outside of the system (two-level approach).

1 Formal computational proofs

1.1 Machines and the quest for correctness

It is generally considered that modern mathematical logic was born towards the end of 19^{th} century, with the work of logicians like Frege, Peano, Russell or Zermelo, which lead to the precise definition of the notion of logical deduction and to formalisms like arithmetic, set theory or early type theory. From then on, a mathematical proof could be understood as a mathematical object itself, whose *correction* obeys some well-defined syntactical rules. In most formalisms, a formal proof is viewed as some tree-structure; in natural deduction for instance, given to formal proofs σ_A and σ_B respectively of propositions A and B, these can be combined in order to build a proof of $A \wedge B$:

$$\dfrac{\dfrac{\sigma_A}{\vdash A} \qquad \dfrac{\sigma_B}{\vdash B}}{\vdash A \wedge B}$$

To sum things up, the logical point of view is that a mathematical statement holds in a given formalism if there exists a formal proof of this statement which follows the syntactical rules of the formalism. A traditional mathematical text can then be understood as an informal description of the formal proof. Things changed in the 1960-ties, when N.G. de Bruijn's team started to use computers to actually build formal proofs and verify their correctness. Using the fact that data-structures like formal proofs are very naturally represented in a computer's memory, they delegated the proof-verification work to the machine; their software Automath is considered as the first *proof-system* and is the common ancestor of today's systems, including Coq. The main motivation for all these systems, apart from the satisfaction of bringing mathematical objects “into life”, is that the non-imaginative mechanical verification of a proof rules out the possibility of errors to a much higher degree than the relying on human understanding.

The arrival of proof systems triggered a new interest in the study of formalisms which were, from then on, judged by their facility of use. Indeed, when one tries to actually build a formal proof in practice, some issues become much more important than when one simply tries to persuade an intelligent reader of the existence of such a proof in principle. For example, an issue like the *size* of the formal proof can become crucial.

1.2 The question of computations

In this particular respect, the subject of computations inside proofs deserves particular attention. In traditional formalisms, *deduction steps* are the only valid way to build a proof. In some cases however, one wishes to also proceed by *calculation steps*. For instance when “proving” that $2 + 2$ is equal to 4.

Typically, addition will be characterized by axioms of lemmas stating that $0+n = n$ and $(n+1)+m = 1+(n+m)$. Even if we omit the steps corresponding to the properties of equality, the proof of $2 + 2 = 4$ will look like

$$\dfrac{\dfrac{\dfrac{\overline{4 = 4}}{0 + 4 = 4}}{1 + 3 = 4}}{2 + 2 = 4}$$

Of course, this approach leads to unreasonably large proofs if one deals with larger numbers.

This issue is particularly acute when dealing with concepts that are highly computational in nature like primality. Since 1951, the largest numbers established as being prime have been so with the use of computer calculations. Of course, this fact is widely accepted. The largest number having been proved prime “by hand” by Ferrier is $(2^{148} + 1)/17$. In other words, if one wants to prove, formally or not, the primality of a number which is not ridiculously small by today standards, one has to somehow incorporate computations into the proof.

Even if it seems clear that proof systems cannot reach the same performances as state-of-the-art dedicated software, one should hope that modern proof systems should be able to go further than what humans could do in 1951. The most natural way to go is to include computations into proofs. Fortunately, modern logical formalisms are often well-equipped for that purpose.

1.3 Conversion rule

The logical formalisms underlying systems like NuPRL, Agda, Alf, Lego, PVS or Coq have an important feature in common. First, the language of objects of these formalisms includes a programming language. In general, this language can be described as the functional core of ML, with some restrictions to ensure termination. In Coq, as in most flavors of type theory, the fact that under assumptions Γ, the object p is a valid proof of proposition P is stated by the sequent $\Gamma \vdash p : P$. Programs, like the addition function, being part of the language, may appear in P, as they do in proposition $2 + 2 = 4$. These programs bear a notion of *evaluation* which corresponds to small-step operational semantics. One writes $=_c$ for the reflexive, symmetric and transitive closure of this relation, which extends to propositions by straightforward congruence.

The fact that programs have a first-class status in these formalisms appears in the logical conversion rule, which, in Coq, reads:

$$\frac{\Gamma \vdash p : A}{\Gamma \vdash p : B} \; (\text{IF} \quad A =_c B)$$

which states that congruent propositions are identified up to the point that they enjoy the same proof-objects. A crucial point is that the calculation steps to go from A to B are not book-kept in the proof object p, as they would be in traditional proofs in, say, first-order logic. This means that lengthy or complex computations may result in proofs that take time to check but remain of manageable size. The idea of computational proofs is thus to exploit this feature by exchanging deduction against computation, aiming at proof objects as small as possible.

As an example, in the case of primality, a very simplified version of computational proofs would be to construct the function which tries to divide a natural number n by all numbers between 2 and $n - 1$. If nat is the type of natural numbers, we have

$$\mathsf{test} : \mathsf{nat} \rightarrow \mathsf{bool}.$$

It is then easy to prove the following correctness "theorem":

$$\forall n : \mathsf{nat}, \mathsf{test}(n) = \mathsf{true} \rightarrow \mathsf{prime}(n).$$

Since $\mathsf{test}(1789)$ evaluates to true, the proposition $\mathsf{test}(1789) = \mathsf{true}$ is proved using the conversion rule and the canonical proof that $\mathsf{true} = \mathsf{true}$. A simple combination with the theorem above yields the proof of $\mathsf{prime}(1789)$.

We can see that through this theorem, we obtain very small primality proofs for any prime number. Of course, these small proofs are uncheckable for large numbers because of the naive algorithm: $\mathsf{test}(n)$ needs too much time to evaluate if n is large. However, following the same idea, we can now build on the tremendous effort made to provide efficient primality checks and import this technology into type theory. Schematically, we only have to provide a cleverer (i.e. faster) version of test.

Safe computations vs. fast computations

The programming language embedded in type theory bears some restrictions since it is primarily meant to be the base of a well-understood logical formalism.

As such, it discards imperative features like mutable variables or arrays. Having to check primality of large numbers with a form of "toy programming language" may look like arbitrary self-restraint. On the other hand, there is a line to draw to decide what computations are safe enough to be considered part of the proof, and which ones lie outside the formal system.

Things are not too bad yet, since the recent versions of Coq benefit from a much faster execution mechanism which uses technology from the programming language world: programs are compiled on-the-fly to a state-of-the-art byte-code. This way to proceed can, we hope, be considered to be a reasonable trade-off between efficiency and safety. With respect to previous execution mechanism using interpreters, one can observe a gain in speed by a factor of 40 and when we write these lines, Coq can probably execute programs much faster than most other comparable proof-systems. Indeed, this work arose largely as an attempt to explore the new possibilities opened by this fast computations in proofs.

Autarkic computation vs. certificates

Once one accepts the dichotomy between fast but unsafe computations outside of the proof-checker and slower computations which are part of the proof-checking process, one has to choose between two possibilities. A first one is to perform all computations inside the system, in an *autarkic* way [3]; this is what happens with the trivial algorithm mentioned above. Another one, is to remark it is easier to find a way out of a labyrinth, when some outside source provides one with an Ariadne's thread (a red line) leading to the exit. In other words, a possibility is to perform some computations outside the system by some dedicated software and then pass-on a *trace* (the red line) of this computation to the proof-checker. This trace gives some information on how computations should be performed and is thus a part of the resulting proof-object. This approach is called *skeptical* computations by Barendregt and was pioneered by Harrison and Théry [8], although in a very different context.

This work is typical for the skeptical computational approach. The idea of building a proof using a *Pocklington certificate* computed by outside means was first used by Caprotti and Oostdijk [6] whose work was the starting base for our effort.

2 Pocklington Certificates

2.1 The theorem

Pocklington's theorem [10] dates back to 1914 and provides a sufficient condition for primality of natural numbers:

Theorem 1. *Given a natural numbers $n > 1$ and a witness a and some pairs $(p_1, \alpha_1), \ldots, (p_k, \alpha_k)$, it is sufficient for n to be prime that the following conditions hold:*

$$\begin{aligned} p_1 \ldots p_k &\ \textit{are prime numbers} && (0)\\ (p_1^{\alpha_1} \ldots p_k^{\alpha_k}) &\mid (n-1) && (1)\\ a^{n-1} &= 1 (\mathsf{mod}\ n) && (2)\\ \forall i \in \{1, \ldots, k\}\ \mathsf{gcd}(a^{\frac{n-1}{p_i}} - 1, n) &= 1 && (3)\\ p_1^{\alpha_1} \ldots p_k^{\alpha_k} &> \sqrt{n}. && (4) \end{aligned}$$

It is worth mentioning, that there is no precisely stated theorem in Pocklington's work. Therefore, the literature often mentions slightly less powerful variants of the previous statement under the same denomination. There are three simple but central observations to make:

- The first one is that, given n, it requires much more computation power to determine suitable numbers $a, p_1, \alpha_1, \ldots, q_k, \alpha_k$ than to check that these numbers verify the conditions 1-4 above. Thus, one says that $a, p_1, \alpha_1 \ldots, p_k, \alpha_k$ form a *Pocklington certificate*. Caprotti and Oostdijk rightly concluded that this was a typical case for the skeptical approach: the certificate is constructed by some outside software and only its verification is done inside the proof system.
- The second observation is that for a natural number n, provided we are given $p_1, \ldots, p_k$ and a, checking primality of n boils down to:
  1. verification of conditions 1-4 which are purely done by numerical computations,
  2. verification of condition 0 which can be done recursively.
- The last observation is that Theorem 1 is the only theorem that needs to be formalized in the prover to insure the correctness of the verification of the certificate, but we also use implicitly its converse: if a number n is prime it is always possible to find a certificate. Given a sufficiently large partial decomposition of $n - 1$, a generator of the multiplicative group $\mathbb{Z}/n\mathbb{Z}$ is a valid candidate for a. Such a generator exists because n is prime so $\mathbb{Z}/n\mathbb{Z}$ is cyclic.

Theorem 1 was the one used by Caprotti and Oostdijk in their experiment described in [6]. Condition 4 indicates that in order to generate a certificate one needs to be able to partially factorize $n - 1$ at least till its square root. For our experiment we are using an improved version proposed by Brillhart, Lehmer and Selfridge in [4]. With this new version, we only need to partially factorize till the cube root. As factorizing $n-1$ is the time-consuming part of finding a certificate, this is a considerable improvement. The theorem that we have formalized in Coq is the following:

Theorem 2. *Given a number n, a witness a and some pairs $(p_1, \alpha_1), \ldots, (p_k, \alpha_k)$ where all the p_i are prime numbers, let*

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

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

$$F_1 \textit{ is even, } R_1 \textit{ is odd, and } F_1R_1 = n-1 \tag{5}$$
$$(F_1+1)(2F_1^2 + (r-1)F_1 + 1) > n \tag{6}$$
$$a^{n-1} = 1(\mathsf{mod}\ n) \tag{7}$$
$$\forall i \in \{1, \dots, k\}\ \mathsf{gcd}(a^{\frac{n-1}{p_i}} - 1, n) = 1 \tag{8}$$
$$r^2 - 8s \textit{ is not a square or } s = 0 \tag{9}$$

The remarks we made about Theorem 1 remain valid for Theorem 2. The existence of a certificate is not direct from Theorem 2 but derives from the exact theorem given in [4] which is a stronger statement:

If conditions 5-8 hold then n is prime iff condition 9 holds.

2.2 The certificates

A certificate is not composed of the a,$p_1, \alpha_1, \dots, p_k, \alpha_k$ alone. In order to be self-contained, it needs also to contain certificates for the p_i's and the factors occurring in these new certificates. This leads to a recursive notion of Pocklington certificate whose verification consists entirely of computations.

This straightforwardly translates to the following recursive definition. A certificate for a single number n is given by the tuple $c = \{n, a, [c_1^{\alpha_1}; \dots; c_k^{\alpha_k}]\}$ where $c_1, \dots, c_k$ are certificates for the prime numbers $p_1, \dots, p_k$. This means certificates can be understood as trees whose branches are themselves certificates corresponding to the prime divisors.

Such structures are easily handled in Coq as an inductive type. A certificate is either:

– such tuples: $c = \{n, a, [c_1^{\alpha_1}; \dots; c_k^{\alpha_k}]\}$[3]
– a pair (n, ψ) composed by a number n and a proof ψ that this number is prime.

The second case is added in order to allow primality proofs which do not rely on Pocklington's theorem. This is useful for 2 (which cannot be proved prime using Pocklington's theorem) but also for using other methods which may be more efficient than Pocklington for some numbers.

Using this representation, a possible certificate[4] for 127 is:

$$\begin{aligned}\{127, 3, [&\{7, 2, [\{3, 2, [(2, \mathsf{prime2})]\}; (2, \mathsf{prime2})]\};\\ &\{3, 2, [(2, \mathsf{prime2})]\};\\ &(2, \mathsf{prime2})]\}\end{aligned}$$

[3] In the following, to shorten certificate we write c_i instead of c_i^1.
[4] This certificate is just an illustration for our purpose, the certificate we automatically generate for 127 is much more concise: $\{127, 3, [\{3, 2, [(2, \mathsf{prime2})]\}; (2, \mathsf{prime2})]\}$.

where prime2 is a proof that 2 is prime. One can remark that this kind of representation duplicates some certificates (here 3 and 2). So, the verification routine will verify many times these certificates. It order to share certificates, we drop trees by flattening them to lists. In this case this yields:

$$[\{127, 3, [7; 3; 2]\}; \{7, 2, [3; 2]\}; \{3, 2, [2]\}; (2, \mathsf{prime2})].$$

We have replaced recursive certificates by their corresponding prime number in decompositions. These certificates appear in the tail of the list. Note that doing so, the certificates for 2 and 3 now appear only once.

This translates straightforwardly into the following Coq definitions:

```
Definition dec_prime := list (positive*positive).

Inductive pre_certif : Set :=
 | Pock_certif : forall n a : positive, dec_prime -> pre_certif
 | Proof_certif : forall n : positive, prime n -> pre_certif.

Definition certificate := list pre_certif.
```

First, we introduce the notion of partial factorization which is a list of prime numbers and their exponent (`dec_prime`). Second, we define the notion of pre-certificates which are either a pair of a prime number n and its primality proof (`Proof_certif`), or a tuple of a prime number n, a witness a and a list of numbers representing a partial factorization of the predecessor of n (`Pock_certif`). This case is not self contained, since it does not contain the primality proofs of the numbers in the partial factorization. This is why we call it pre-certificate.

A complete certificate is a list of pre-certificates. The head of the list is generally a triple (`Pock_certif` n a d), the tail contains pre-certificates for numbers appearing in the factorization list d of the first one (and recursively).

3 Checking certificates

Since this last definition is basically a free structure, the existence of an object of type `certificate` does not yet allow to state anything about primality. We focus on the function C which verifies the validity of a certificate and on the correctness proof of this function.

Our goal is to define C as a function from certificates to boolean, returning true if the list implies the primality of the numbers it contains. Remark that, for our purpose, the completeness of the function C is not required; the correctness lemma we want to prove is:

$$\mathsf{Pock\_refl} : \forall c, l,\ C\ (c :: l) = \mathit{true} \Rightarrow \mathsf{prime}\ (n\ c)$$

where $(n\ c)$ is the prime number contained in the pre-certificate c. In fact, we prove a more general lemma:

$$\forall l,\ C\ l = \mathit{true} \Rightarrow \forall c \in l,\ \mathsf{prime}\ (n\ c)$$

The function C is defined recursively over the list l. If the list l is empty, the function returns true. If the list is not empty ($l = c :: l'$), the function performs a recursive call over l', this implies the validity of all the pre-certificates in l' and so the primality of their associated numbers. Then the function verifies the validity of the pre-certificate c, there are two cases:

– If c is given by its proof form $c = (n, \psi)$, in that case there is nothing to do (the type checking of Coq ensures that ψ is a valid primality proof for n).
– If c is a Pocklington pre-certificate $c = \{n, a, [p_1^{\alpha_1}; \dots ; p_k^{\alpha_k}]\}$. The function first verifies that all the divisors $p_1, \dots, p_k$ have a corresponding pre-certificate in the list l' (this implies that all the divisors are prime). If this is the case, the function checks that the pre-certificate verifies conditions 5-9; this is done by another function C_c.

Checking the computational conditions

The function C_c starts by computing the numbers F_1, R_1, s, r as defined in theorem 2. Verifying conditions 5 and 6 is then straightforward. For conditions 7 and 8 the difficulty is to efficiently compute $a^{n-1} \bmod n$ and $\mathsf{gcd}(a^{\frac{n-1}{p_i}} - 1, n)$ for $i = 1 \dots k$. It is important not to compute a^{n-1} and $a^{\frac{n-1}{p_i}}$, which can be huge. We do this by always working modulo n since $\mathsf{gcd}(b, n) = \mathsf{gcd}(b \bmod n, n)$. Furthermore, we can compute only one gcd using the fact that $\mathsf{gcd}(b_1 \dots b_l, n) = 1$ iff for all $i = 1 \dots l$, $\mathsf{gcd}(b_i, n) = 1$.

We define the following functions working modulo n:

– a predecessor function (`Npred_mod`);
– a multiplication function (`times_mod`);
– a power function (`Npow_mod`) (using the *repeated square-and-multiply algorithm*);
– a multi-power function (`fold_pow_mod`) that given a, $l = [q_1; \dots ; q_r]$ and n computes $a^{q_1 \cdots q_r} \bmod n$.

Another optimization is to share parts of the computations of

$$a^{\frac{n-1}{p_1}} \bmod n, \; \dots, \; a^{\frac{n-1}{p_k}} \bmod n, \; a^{n-1} \bmod n.$$

Let $m = (n-1)/(p_1 \dots p_k)$, if we perform these computations separately, $a^m \bmod n$ is computed $k+1$ times, $(a^m \bmod n)^{p_1} \bmod n$ is computed k times, and so on.

To share computation we define the following function:

```
Fixpoint all_pow_mod (P A : N) (l:list positive) (n:positive)
                                              {struct l}: N*N :=
  match l with
  | nil => (P,A)
  | p :: l =>
    let m := Npred_mod (fold_pow_mod A l n) n in
    all_pow_mod (times_mod P m n) (Npow_mod A p n) l n
  end.
```

If P and A are positive numbers less than n and l the list $[q_1; \ldots ; q_r]$, the function `all_pow_mod` returns the pair:

$$(P \prod_{1 \leq i \leq r} A^{\frac{q_1 \cdots q_r}{q_i}} \ \mathsf{mod}\ n, A^{q_1 \cdots q_r} \ \mathsf{mod}\ n)$$

Remark that the application of this function to $P = 1$, $A = a^m \ \mathsf{mod}\ n$ and $l = [p_1; \ldots ; p_k]$ leads to the numbers we need to compute. Note that the order of the list l is important for efficiency. A^{p_1} is computed only once, but power of elements in the tail of the list are computed more than once. So, the function is more efficient if l in sorted in decreasing order.

Finally, the function C_c checks the condition 9 ($s = 0 \lor r^2 - 8s$ is not a square). If $s \neq 0$ and $r^2 - 8s \geq 0$ it should check that $r^2 - 8s$ is not a square. To do so, we slightly adapt the definition of pre-certificates by adding the low integer square root[5] sqrt, and we only verify that

$$\mathsf{sqrt}^2 < r^2 - 8s < (\mathsf{sqrt} + 1)^2$$

To sum up the extended inductive type for pre-certificate is

```
Inductive pre_certif : Set :=
 | Pock_certif : forall n a sqrt: positive, dec_prime -> pre_certif
 | Proof_certif : forall n : positive, prime n -> pre_certif.
```

Defining arithmetic operations in Coq

The programming language of Coq is functional. Datatypes can be defined using inductive types. One can then write recursive functions over datatypes using structural recursion only. This restriction and the type system ensure the strong normalization of the language from which the logical soundness of the system is derived.

When doing computational proofs, each time we define a function we also need to prove formally its correctness. This means that most of the time we have to do a compromise between the efficiency of the function, the difficulty of writing it in a structural recursive functional language and the cost of proving its correctness.

For verifying Pocklington certificate, we need basic integer operations. Instead of starting from scratch, we have decided to use the standard library of Coq for integer arithmetic. In this library, positive numbers are represented by a list of bits as follows:

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

[5] In the case where $r^2 - 8s < 0$ we put a dummy number 1, since $r^2 - 8s$ is trivially not a square.

`xH` represents 1, `xO` x represents $2x$, `xI` x represents $2x+1$. So 6 is represented by `(xO (xI xH))`. All the operations are purely functional (i.e. no machine arithmetic and no side effect). This means that every time an operation returns a number, this number has to be allocated. In our application, allocating and garbage collecting are particularly time consuming.

To minimize allocation we had to re-implement some operations. For example, in the standard library the remainder function is defined as the second component of the euclidean division. This means that computing a remainder also allocates for the quotient. We provide a more direct implementation. Also we have implemented a special function for squaring numbers.

4 Building certificates

The goal of this section is to explain how to build certificates for large prime numbers. The main difficulty is to compute a partial factorization of the predecessor of the prime n we want to prove.

As said above, this factorization is naturally done relying on tools outside of the proof system. The software building the certificate thus plays the role of an oracle whose prediction is however carefully verified. We built this software as a C program based on the ECM library [1] and the GMP library [2]. Given a number n, the program generates Coq files whose lemmas have the following form:

```
Lemma prime_n : prime n.
Proof.
 apply (Pock_refl (Pock_certif n a d sqrt) l).
 exact_no_check (refl_equal true).
Qed.
```

where a is the witness of the Pocklington theorem, d a partial factorization of $n-1$, sqrt the square root of r^2-8s (if r^2-8s is positive otherwise 1) and l is the list of pre-certificates proving that all numbers in the factorization are prime. The proof starts by an application of the reflexive theorem Pock_refl. At proof-checking time, the system checks that $C((n,a,d,\mathsf{sqrt}) :: l) = \mathsf{true}$ or, to be more precise, that this proposition is convertible with the proposition true = true. Indeed, refl_equal true simply stands for the canonical proof of true = true.

This last step is really the reflexive part of the proof, where deduction is replaced by computation. As we can see, the computation step does not appear in the proof which allows to build very small proofs. In our case, the size of the proof is mainly the size of the certificate $(n,a,d,\mathsf{sqrt}) :: l$.

A technical detail for Coq users: during the construction of the proof, we ask the system not yet to check that $C((n,a,d,\mathsf{sqrt}) :: l) = \mathsf{true}$ is convertible with true = true. This is done only once at the validation step (Qed). The ability of the proof checker to compute $C((n,a,d,\mathsf{sqrt}) :: l)$ is crucial, for the verification time.

4.1 Generating certificates for arbitrary numbers

The difficult task for the oracle is to obtain the partial factorization and to find the witness. The basic options of the oracle are the following:

```
pocklington [-v] [-o filename] [ prime | -next num ]
```

`pocklington` is the oracle that generates a certificate for the prime number *prime* or the next prime number following the number *num*. The `-v` option is for verbose mode, `-o` allows to chose the name of the output file, if none is given a default name is created depending of the prime number.

The oracle first checks that the number n has a large probability to be prime, and then tries to find a pseudo decomposition of its predecessor. To compute this decomposition it first tries to get all the small factors of n using trivial divisions by 2, 3, 5 and 7 and then by all the numbers greater than 7 that are not a multiples of 2, 3, 5 and 7. The division limit is the maximum of one million and $\log_2(n)^2$. Then, if the pseudo decomposition obtained so far is still less than the cubic root of n, the oracle tries to find larger factors using the ECM library.

The ECM library proposes three methods to find factors: (p-1, p+1 and ecm). We have developed a simple heuristic that successively calls the three methods to find, first, factors of less that 20 digits, then, less that 25 digits and so on up to factors less than 65 digits. This process is not deterministic and not exhaustive. The iteration terminates as soon as we get a sufficiently large partial decomposition. So, we can miss some factors and we are not sure that the process terminates in a reasonable time.

When a partial decomposition has been found, the oracle tries to find the witness a for the Pocklington criteria by testing 2, 3, 4 and so on. Finally it calls itself recursively on the numbers in the decomposition for which no certificate has yet been generated.

In order to share computations, the certificates for the first 5000 primes (form 2 to 48611) are provided at the start in a separated file `BasePrimes.v`. Thus certificates for prime numbers less than 48611 are not regenerated.

Using this technique, the oracle is able to generate certificates for most of the prime numbers containing less that 100 digits (in base 10), and for some larger primes.

4.2 Generating certificates for Mersenne numbers

Mersenne numbers are those of the form $2^n - 1$. Not all of them are prime. The first ones are for $n = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127$. Until now only 43 are known to be prime and it is still open whether there are infinitely many or not. Currently, the last one has been found by Curtis Cooper and Steven R. Boone on December 15, 2005. $2^{30402457} - 1$ is the new largest known prime number and has 9152052 digits!

The first striking remark is that such a number is written $1111\ldots1111$ in base 2. The second is that the decomposition of its predecessor always contains

2, since $(2^n - 1) - 1 = 2(2^{n-1} - 1)$. So, finding a decomposition of its predecessor is equivalent to finding a decomposition of $2^{n-1} - 1$.

Here we can use some very simple arithmetical properties to start the decomposition:

- $2^{2p} - 1 = (2^p - 1)(2^p + 1)$
- $2^{3p} - 1 = (2^p - 1)(2^{2p} + 2^p + 1)$

The oracle uses those tricks recursively to start the decomposition of the predecessor of Mersenne number, this allows to considerably reduce the size of the numbers to factorize. Since $2^n - 1$ can be prime only if n is odd, we know $n - 1$ is even and the first remark always applies. When we reach the point where these two remarks do not apply any more, the oracle uses the method for arbitrary numbers to factorize the resulting numbers as described above.

The syntax to compute the certificate for the Mersenne number $2^n - 1$ is the following: `pocklington -mersenne` *n*. This technique allows to build certificates for the 15 firsts Mersenne numbers, the largest one is for $n = 1279$ and has 386 digits.

When n grows further the resulting numbers are too big for their factorizations to be computed. So we have added a new entry to the oracle that takes as argument a file containing a prime number and a partial decomposition of its predecessor. The syntax is the following: `pocklington -dec` *file*.

We can use the trick described below and the tables [5] to build the file. These tables contain most of the factorizations of $b^n \pm 1$ for $b = 2, 3, 5, 6, 7, 10, 11, 12$ and n a high power (for $b = 2$, n should be less than 1200). Those tables are available directly on the Web[6]. Using these techniques we have been able to compute a certificate for 16th ($n = 2203$) and the 17th ($n = 2281$) Mersenne numbers.

4.3 Going to the limit

Some similar problems appears for the 18th Mersenne number ($n = 3217$): using the table we are able to compute the complete factorization of

$$2^{1608} - 1 = (2^{804} - 1)(2^{804} + 1)$$

but we are not able to find a certificate for one of the prime divisors, which is a number of only 90 digits.

To solve this, our idea is to find some prime divisors of $2^{1608} + 1$ big enough to replace the prime divisor of 90 digits in the pseudo decomposition. The problem is that $2^{1608} + 1$ is not in the table. To factorize $2^{1608} + 1$ we used a well known property of cyclotomic polynomial[7]:

$$x^n + 1 = (x^{2n} - 1)/(x^n - 1) = \prod_{d|2n} \Phi_d(x) / \prod_{d|n} \Phi_d(x)$$

[6] http://homes.cerias.purdue.edu/ ssw/cun/prime.php

[7] a complete explanation can be found in [9]

where $\Phi_d(x)$ is the cyclotomic polynomial, given by:

$$\Phi_d(x) = \prod_{\delta | d} (x^\delta - 1)^{\mu(d/\delta)}$$

where μ is the Möbius function [9].

Setting $2n = 2^t m$ with m odd, leads to:

$$x^n + 1 = \prod_{d|m} \Phi_{2^t d}(x)$$

where all the $\Phi_{2^t d}(x)$ are divisors of $x^n + 1$ that we can factorize.

Using this trick we have been able to compute enough prime divisors of $2^{1608} + 1$ to replace the 90 digits prime divisors of $2^{1608} - 1$, and to build certificates for all its prime divisors. Finally this allowed us to build a certificate for the 18th Mersenne number which yields 969 digits. This number was first proved prime by Riesel in 1957.

Note that Pocklington certificate is known not to be the most efficient way to verify the primality of Mersenne numbers. Lucas test gives a much simpler criterion since it does not require any factorization:

Theorem 3. *Let (S_n) be recursively defined by $S_0 = 4$ and $S_{n+1} = S_n^2 - 2$, for $n > 2$, $2^n - 1$ is prime if and only if $(2^n - 1)|S_{n-2}$.*

We have also formalized this theorem in our prover to compare the running time of this test with our generated Pocklington certificates for Mersenne numbers.

To do so we add a new entry `Lucas_certif` in our inductive type:

```
Inductive pre_certif : Set :=
 | Pock_certif : forall n a : positive, dec_prime -> pre_certif
 | Proof_certif : forall n : positive, prime n -> pre_certif
 | Lucas_certif : forall n p : positive, pre_certif.
```

where n should be $2^p - 1$.

To generate certificates for Mersenne numbers using Lucas test, we add a new entry to the oracle: `pocklington -lucas` p.

4.4 Generating certificate for Proth numbers

A more friendly set of prime numbers for Pocklington certificate are the Proth numbers. They are numbers of form $k2^p + 1$, where k is odd.

Providing that k is sufficiently small with respect to 2^p, the partial decomposition reduces to 2^p. So the generation of the corresponding certificate is trivial.

To generate Pocklington certificates for Proth number we add a new entry to the oracle: `pocklington -proth` k p.

5 Performances

All the certificates described in the section above have been checked in Coq. The software and formal material are available at the following location:

`http://www-sop.inria.fr/everest/Benjamin.Gregoire/primnumber.html`

Certificates are generated by the pocklington program that is composed of 1721 lines of C code. They are then checked in Coq using our library that consists of 6653 lines of proof script. We provide some performance figures for the cvs version 8.0 of Coq with processor Intel Pentium 4 (3.60 GHz) and a RAM of 1Gb. Of course, this version of Coq uses the compilation scheme described in [7].

from - to	build	size	verify
2 - 5000	0.15s	989K	35.85s
5001 - 10000	0.17s	1012K	42.59s
10001 - 20000	0.38s	2.1M	134.14s
20001 - 30000	0.38s	2.1M	138.30s
30001 - 40000	0.38s	2.1M	145.81s
40001 - 50000	0.38s	2.2M	153.65s
50001 - 60000	0.41s	2.2M	153.57s
60001 - 70000	0.43s	2.2M	158.13s
70001 - 80000	0.39s	2.2M	160.07s
80001 - 90000	0.40s	2.2M	162.58s
90001 - 100000	0.44s	2.2M	162.03s

Fig. 1. Time to verify the first 100000 prime numbers

Figure 1 gives the time to build the certificates for the 100000 first primes, the size of the certificate and the time for the Coq system to check them. On average, generating certificates for a small prime number takes about 3.10^{-5} seconds, their sizes are 215 bytes average, and it takes about 0.0144 seconds to verify.

		size		time		
prime	digits	deduc.	refl.	deduc.	refl.	refl. + VM
1234567891	10	94K	0.453K	3.98s	1.50s	0.50s
74747474747474747	17	145K	0.502K	9.87s	7.02s	0.56s
1111111111111111111	19	223K	0.664K	17.41s	16.67s	0.66s
$(2^{148}+1)/17$	44	1.2M	0.798K	350.63s	338.12s	2.77s
P_{200}	200	_	2.014K	_	_	190.98s

Fig. 2. Comparison with the non recursive method

Figure 2 makes a comparison between the deductive approach (the one developed by Oostdijk and Caprotti) and our reflexive one, using curious primes. The 44 digits prime number $(2^{148}+1)/17$ is the biggest one proved in Coq before our work.

As expected, the reflexive method considerably reduces the size of the proof, as showed by the size column of the table. For the prime number $(2^{148}+1)/17$, the size is reduced by a factor 1500. The reduction of the proof size is natural since explicit deduction is replaced by implicit computation.

The three last columns compare the verification time for the different approaches. Without the use of the virtual machine, the reflexive approach is a little bit faster, but the times are comparable. If we verify the proof with the version of Coq using a virtual machine to perform the conversion test (here to compute the result of the C function), we can observe a gain of a factor 9 for small examples to more than 120 for the biggest ones. This means that the combination of computational reflexion with the virtual machine allows small proofs that are quickly verified.

Note that when verifying a reflexive proof the time consuming task is the reduction of decision procedure. It is precisely the reduction that the virtual machine improves. This explains why we get such a speed up. Using virtual machine for checking deductive proofs usually does not provide any speed up since these proofs do not use much reduction.

Using the reflexive approach, we have been able to prove a new random prime number of 200 digits:

$$\begin{aligned}P_{200} = \ & 67948478220220424719000081242787129583354660769625 \\ & 17084497493695001130855677194964257537365035439814 \\ & 34650243928089694516285823439004920100845398699127 \\ & 45843498592112547013115888293377700659260273705507\end{aligned}$$

in 191 seconds and the proof size is 2K. In practice, it is difficult to find factors of more than 35 digits. Most numbers with less than 100 digits contain sufficient factors of less than 20 digits, so ECM finds them rapidly. For larger numbers, being able to generate a certificate for N is a question of luck, $N-1$ must be smooth (i.e. it contains a lot of small factors). This is the case for P_{200}.

Figure 3 gives the time to verify the certificates of the 18th first Mersenne numbers (using a Pocklington certificate), except for the 7th first which are part of the first 100000 primes. The biggest one is a 969 digit number. We have not been able to generate certificates for numbers greater than the 18th. For the 19th Mersenne number, we have not been able to find a sufficiently large partial factorization of its predecessor. For the 20th, we have such a factorization (using the table of aurifeuillian polynomial from [5]), but we are not able to recursively generate the certificates for two of its factors.

For Mersenne numbers, we can do better than using a Pocklington certificate using Lucas test. The last column gives the time to verify in Coq the 20 first Mersenne number using this test. As far as we know, the 20th Mersenne number is the biggest prime number that has been formally proved in a proof assistant.

#	n	digits	years	discoverer	certificate	time	time(Lucas)
8	31	10	1772	Euler	0.527K	0.51s	0.01s
9	61	19	1883	Pervushin	0.648K	0.66s	0.08s
10	89	27	1911	Powers	0.687K	0.94s	0.25s
11	107	33	1914	Powers	0.681K	1.14s	0.44s
12	127	39	1876	Lucas	0.775K	2.03s	0.73s
13	521	157	1952	Robinson	2.131K	178.00s	53.00s
14	607	183	1952	Robinson	1.818K	112.00s	84.00s
15	1279	386	1952	Robinson	3.427K	2204.00s	827.00s
16	2203	664	1952	Robinson	5.274K	11983.00s	4421.00s
17	2281	687	1952	Robinson	5.995K	44357.00s	4964.00s
18	3217	969	1957	Riesel	7.766K	94344.00s	14680.00s
19	4253	1281	1961	Hurwitz	_	_	35198.00s
20	4423	1332	1961	Hurwitz	_	_	39766.00s

Fig. 3. Time to verify Mersenne numbers

For all these benchmarks, we have to keep in mind that Coq uses its own arithmetic: numbers are encoding by a inductive type, i.e. a chained list of booleans. No native machine arithmetic is used. This means that the computations which are done to check certificates are more similar to symbolic computation than to numerical computation. For example, when dealing with the 20th Mersenne number, a 4422 digits number in base 2 (1332 in base 10), we manipulate list of 4423 elements. The fact that we are capable to perform such a symbolic computation clearly indicates that the introduction of the virtual machine in Coq is an effective gain in computing power.

6 Conclusion

Proofs are what ensures the correctness of computations, but many results of mathematics or computer science can only be established through computations. It does even seem likely that the proportion of computationally obtained results will grow in the future. But the more intricated computation and deduction become, the more difficult it is to have a global overview of the resulting construction; this raises a serious question regarding the reliability of the final statement. This issue is important, especially when considering that important mathematical results now rely, at least partly, on computations. Famous examples are the four color theorem or the Kepler conjecture.

The ultimate goal of works as the present one is to establish proof systems as the place where proofs and computations interleave in a fruitful and yet safe way. In the case of Coq, recent progress in the implementation of reduction was a significant step in reducing the overhead for doing computations inside the proof system rather than outside. Even if progress should still be made in that respect, it appears that what we have been able to prove, for typically computational statements, is not ridiculous and represents a step in the right direction.

In order to go further on the topic of formal primality proofs, two ways should obviously be explored. A first one is to formalize more modern primality proofs, as the ones relying on algebraic curves. This means translating to Coq an interesting body of mathematics. A second direction is to pursue the work on the system itself. Indeed, a current mechanical bottleneck is certainly the very inefficient representation of numbers in the proof system (basically as lists of bits). Integrating numbers into type theory with a more primitive status, allowing operations which could be implemented using more low-level features of the processor and doing this in a safe way is an interesting challenge and would be a further step in the quest of computationally efficient proof systems. Once this done, we can hope for a powerful system for combining deduction, symbolic computation and numerical computation in a safe and integrated way.

Acknowledgments Thanks are due to Henk Barendregt who initially motivated this work, to Paul Zimmermann for answering all our arithmetic questions and for developing the wonderful ECM library. Finally, we want to thank our anonymous referees for their stimulating comments.

References

1. *Elliptic Curve Method Library.* http://www.loria.fr/˜ zimmerma/records/ecmnet.html.
2. *GNU Multiple Precision Arithmetic Library.* http://www.swox.com/gmp/.
3. H. Barendregt and E. Barendsen. Autarkic computations in formal proofs. *J. Autom. Reasoning*, 28(3):321–336, 2002.
4. J. Brillhart, D. H. Lehmer, and J. L. Selfridge. New primality criteria and factorizations of $2^m \pm 1$. *Mathematics of Computation*, 29:620–647, 1975.
5. J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman, and S. S. Wagstaff, Jr. *Factorizations of* $b^n \pm 1$, volume 22 of *Contemporary Mathematics.* American Mathematical Society, Providence, R.I., 1983. $b = 2, 3, 5, 6, 7, 10, 11, 12$ up to high powers.
6. O. Caprotti and M. Oostdijk. Formal and efficient primality proofs by use of computer algebra oracles. *Journal of Symbolic Computation*, 32(1/2):55–70, July 2001.
7. B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.
8. J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *J. Autom. Reasoning*, 21(3):279–294, 1998.
9. S. Lang. *Algebra*, volume 211 of *Graduate Texts in Mathematics.* Springer-Verlag, New York, third edition, 2002.
10. H. C. Pocklington. The determination of the prime or composite nature of large numbers by Fermat's theorem. volume 18, pages 29–30, 1914.