Formal and Efficient Primality Proofs by Use of Computer Algebra Oracles

OLGA CAPROTTI† AND MARTIJN OOSTDIJK‡

†RISC-Linz Research Institute for Symbolic Computation, Johannes Kepler University,
A-4020 Linz, Austria
‡Department of Mathematics and Computer Science, Eindhoven University of
Technology, P.O. Box 513, 5600MB, Eindhoven, The Netherlands

This paper focuses on how to use Pocklington’s criterion to produce efficient formal
proof-objects for showing primality of large positive numbers. First, we describe a formal
development of Pocklington’s criterion, done using the proof assistant Coq. Then we
present an algorithm in which computer algebra software is employed as oracle to the
proof assistant to generate the necessary witnesses for applying the criterion. Finally, we
discuss the implementation of this approach and tackle the proof of primality for some
of the largest numbers expressible in Coq.

© 2001 Academic Press

1. Introduction

The problem of showing whether a positive number is prime or composite is historically
recognized to be an important and useful problem in arithmetic. Since Euclid’s times,
the interest in prime numbers has only been growing. For today’s applications, primality
testing is central to public key cryptography and for this reason is still heavily investigated
in number theory (Ribenboim, 1996).

Although the problem is clearly decidable, the trivial algorithm, derived from the defi-
nition, that checks for every number \( q \) such that \( q \leq \sqrt{n} \) whether \( q \mid n \), is far too inefficient
for practical purposes. There exist several alternative methods to check primality, how-
ever in this paper we deal exclusively with proofs obtained by a classical criterion due
to Pocklington in 1914 (Pocklington, 1914). Our interest is motivated by the fact that
in order to produce a proof of primality the criterion needs to find numbers that verify
certain algebraic equalities. These numbers are easily generated by a computer algebra
oracle, for instance the systems GAP, Magma, or Pari (Schöner et al.; Batut et al., 2000;
The Magma Computational Algebra System, 2001) deal well with number theoretical
questions. From the computer algebra point of view, it has already been shown how an
informal textual proof of primality generated by GAP can be turned into an interactive
mathematical document (Caprotti and Cohen, 2001) in natural language in which the
algebraic assertions can be interactively verified using a computational back-engine. In-
stead, the approach of this paper is from the theorem-proving point of view and the main
problem is efficiently constructing a formal, verifiable proof-object (a lambda term) rep-
resenting a proof of primality. This proof-object can then be visualized as an interactive

§During this work, the author was supported by the OpenMath Esprit project 24969.

Notice that the cooperation of theorem provers with computer algebra is essential for being able to solve this task. Theorem provers are very limited on the amount and type of computations they can perform (Barendregt and Barendsen, 2001), however, they are very well suited for organizing the logical steps of a proof. On the other hand, although computer algebra systems have algorithms for deciding whether a number is prime or not, it is hard for them to produce a proof of primality, let alone one that is formally verifiable by a theorem prover. Additionally, in certain systems, the algorithms employed to test primality (or compositeness) of large primes behave probabilistically and there is a slight possibility that the returned answer is not true. Thus, the winning strategy is to combine both kinds of system. There is a wide amount of literature both on the subject of combining systems for problem solving and on specific case studies, among many (see for instance Clarke and Zhao, 1992; Calmet and Homann, 1996; Giunchiglia, 1996; Buchberger et al., 1997; Calmet and Homann, 1997; Bertoli et al., 1998; Dunstan, 1998).

Computer algebra has been incorporated in theorem proving following three approaches, based on the level of trust, that range from believing, through skeptical, to autarkic (Harrison and Théry, 1998; Barendregt and Barendsen, 2001; Cohen and Barendregt, 2001). The believing approach trusts completely the results given by a computer algebra system and treats them as axioms (Ballarin et al., 1995). The skeptical approach does not trust completely the results given by a computer algebra oracle and treats them as witnesses that require formal verification (Harrison and Théry, 1993; Kerber et al., 1998; Sorge, 2000). This approach presupposes that the chosen oracle is good at finding the witnesses. The autarkic approach refuses to consult any oracle and does all the computations inside the theorem prover.

This paper extends the initial work presented in Caprotti and Oostdijk (2000) on how Pocklington’s criterion can be employed in a skeptical approach to produce eficient formal proof-objects that show primality of large positive numbers in a proof assistant such as Coq (Projet Coq, 1999). The claim of formality relies on the complete development and proof of Pocklington’s criterion within Coq which has been restated in a variant that allows use of integers instead of natural numbers. The claim of efficiency† is sustained by the fact that we have been able, for instance, to construct and verify the proof of primality of Ferrier’s prime, a 44-digits long prime number (the longest one found before the computer age) arbitrarily chosen among the ones listed in Honaker (2000). The implementation is geared toward the possibility of switching to a different computer algebra engine if need arises: for instance when factorization of large integers is required the user may easily choose a more powerful engine as oracle. Additionally, the user may define at which level of belief the proof should be produced, from total trust in each algebraic result coming from the oracle to total skepticism.

The structure of this paper is as follows. The formalization of Pocklington’s criterion is described in Section 2. Section 3 gives the architectural details for using it to produce formal proofs of primality with the aid of computer algebra oracles. Our implementation of the algorithm and timings for some benchmarks are discussed in Section 4 and the paper’s conclusions are in Section 5.

†Intended in the context of theorem proving.
2. Formalization

The Pocklington criterion is one of many number theoretical results that are useful for verifying primality of a positive number $n$, see for instance Ribenboim (1996) for more examples. This section describes the development of a formal proof of the criterion in COQ, starting with the informal proof. Such a formalization is a prerequisite necessary before producing primality proof-objects that can be automatically checked for correctness. The work presented in this section is a follow-up of Elbers (1998), where the use of Pocklington’s Criterion in a formal setting was first investigated.

**Lemma 2.1. (Pocklington’s Criterion)** Let $n \in \mathbb{N}$, $n > 1$ with $n - 1 = q \cdot m$ such that $q = q_1 \cdots q_t$ for certain primes $q_1, \ldots, q_t$. Suppose that $a \in \mathbb{Z}$ satisfies $a^{n-1} \equiv 1 \pmod{n}$ and $\gcd(a^{\frac{n-1}{q_i}} - 1, n) = 1$ for all $i = 1, \ldots, t$. If $q \geq \sqrt{n}$, then $n$ is a prime.

**Proof.**

1. Let $p | n$ and $\text{Prime}(p)$, put $b = a^m$.
2. Then $b^q = a^{mq} = a^{q-1} = 1 \pmod{n}$.
3. So $b^q = 1 \pmod{p}$.
4. Now $q$ is the order of $b$ in $\mathbb{Z}^*_p$, because:
5. Suppose $b^\frac{n}{q_i} = 1 \pmod{p}$, then $a^{\frac{n}{q_i}} = a^{\frac{n-1}{q_i}} = 1 \pmod{p}$.
6. There exist $\alpha, \beta \in \mathbb{Z}$ such that $\alpha(a^{\frac{n-1}{q_i}} - 1) + \beta n = 1 \pmod{p}$.
7. So, $\alpha(1 - 1) + \beta 0 = 1 \pmod{p}$. Contradiction.
8. By Fermat’s little theorem: $b^{p-1} = 1 \pmod{p}$, therefore $q \leq p - 1$, so $\sqrt{n} \leq q < p$.
9. Hence for every prime divisor $p$ of $n$: $p > \sqrt{n}$.
10. Therefore $\text{Prime}(n)$. □

Although the proof is easy from a mathematical viewpoint, the exercise of formalizing it in COQ is not a straightforward one.

The COQ system is a logical system based on typed lambda calculus (Barendregt, 1992). Propositions can either be assumed as axioms, or they can be proven by constructing a formal proof-object. A complete formalization of Pocklington’s criterion amounts to finding a proof-object, i.e. a lambda term, which inhabits the type corresponding to the statement of the criterion.

Our development of the criterion is a full formalization, meaning that no lemma is assumed as axiom. Building the formal proof from the basic lambda primitives requires a lot of work. However, the COQ proof assistant is equipped with a library of defined concepts concerning standard mathematical theories including natural numbers, integers, relations, and lists. Furthermore, it has a powerful language of tactics which allows the user to specify proofs as abstract tactic scripts instead of concrete lambda terms.

Formalization starts with identifying those mathematical concepts used in the proof that are not yet in the standard library. Many notions that are part of the repertoire of any mathematician are not (yet) in the standard library. The most prominent of the concepts necessary are division and primality on the naturals, equality modulo $n$, greatest common divisor, exponentiation, and the order of an element $b$ in the multiplication group $\mathbb{Z}^*_p$. They are formalized in the natural way by the following definitions.
Definition \text{Divides}(n, m) = \exists q \in \mathbb{N}. (m = n \cdot q)

Definition \text{Prime}(n) = (n > 1) \land \forall q \in \mathbb{N}. (q \mid n \rightarrow (q = 1 \lor q = n))

Definition \text{Mod}(a, b, n) = \exists q \in \mathbb{Z}. (a = b + n \cdot q)

Definition \text{Gcd}(a, b, c) = c \mid a \land c \mid b \land \forall d \in \mathbb{N}. ((d \mid a \land d \mid b) \rightarrow (d \leq c))

Fixpoint \begin{align*}
\text{Exp}(a, 0) &= 1 \\
\text{Exp}(a, n + 1) &= a \cdot \text{Exp}(a, n)
\end{align*}

Definition \text{Order}(b, q, p) = (0 < q) \land b^q = 1 \pmod{p} \land \forall d \in \mathbb{N}. (((0 < d) \land b^d = 1 \pmod{p})) \rightarrow (q < d).

The definition of \text{Exp} is a so-called fixpoint definition, which means that \text{Exp} is defined by well-founded recursion. In practice this means that \text{Exp}(a, n) is convertible to \(a^n\) for concrete values of \(a\) and \(n\), and for example finding an inhabitant for the statement \text{Exp}(2, 3) = 8 is as easy as finding an inhabitant for \(8 = 8\). The definitions of \text{Divides} and \text{Mod} do not have this computational behavior, so in order to prove for example \text{Divides}(2, 8), one has to provide the witness \(q = 4\). For brevity \text{Divides}(n, m) is denoted as \(n \mid m\), \text{Mod}(a, b, n) is denoted as \(a = b \pmod{n}\), and \text{Exp}(a, n) is denoted as \(a^n\).

The definition of concepts alone is not enough. Many trivial (and less trivial) lemmata about the concepts have to be proven so that they can be used in the course of proving the criterion. The definitions together with the lemmata are grouped together in COQ modules representing mathematical theories. In this way, the theories can be reused when formalizing other parts of mathematics, for instance similar criteria. Figure 1 gives an overview of the different modules developed to prove Pocklington’s criterion. For the modules that are not part of the standard library, the size is given. The modules are COQ vernacular files and are available online (Pocklington development files, 2000).
The Arith and ZArith modules are provided by Coq to support basic arithmetic on the natural and integer numbers respectively. The natural numbers are implemented inductively with constructors for zero element and successor function. This unary representation makes these natural numbers very inefficient for computation on concrete instances. The integer numbers are also implemented inductively but with a binary representation which is much more suited for concrete computations. However, when reasoning about abstract numbers, for example to prove results by induction, the binary representation can become a hindrance. To overcome these difficulties, Coq results are available that allow conversion between the slow naturals and the fast integers. We developed some more in the natZ module. This allows switching between the two representations in the proof of the criterion. When the criterion is applied in the next section to generate concrete primality proofs, the binary representation is used exclusively.

The natZ module is part of a layer of modules built on top of the arithmetic modules. This layer develops some mathematical tools, it consists of data structures and lemmata to allow a slightly higher level of reasoning later on. The module lemmas collects the additional lemmata on the elementary arithmetic which was needed during the development. The theory of finite lists is in the list module; it is heavily used in reasoning about prime factorization and in the proof of Fermat’s little theorem. The dec theory contains lemmata useful for proving decidability of predicates in general. Decidability of a predicate $P$ in the context of constructive theorem provers like Coq means that the principle of the excluded middle, $P(n) \lor \lnot P(n)$, holds. One may carry out a formalization in Coq in classical logic by assuming the principle of the excluded middle as an axiom holding for any proposition. Instead, our formalization of Pocklington’s criterion is done fully constructively. The dec module starts with some decidability proofs for simple predicates.

\begin{align*}
\text{Lemma eqdec: } & \forall n, m : \mathbb{N}. (n = m \lor \lnot (n = m)) \\
\text{Lemma ledec: } & \forall n, m : \mathbb{N}. (n \leq m \lor \lnot (n \leq m)) \\
\text{Lemma ltdc: } & \forall n, m : \mathbb{N}. (n < m \lor \lnot (n < m)) \\
\text{Lemma gedc: } & \forall n, m : \mathbb{N}. (n \geq m \lor \lnot (n \geq m)) \\
\text{Lemma gtdec: } & \forall n, m : \mathbb{N}. (n > m \lor \lnot (n > m)).
\end{align*}

Other lemmata in dec serve as tools for proving decidability for compound predicates. Decidability is preserved by the propositional connectives.

\begin{align*}
\text{Lemma notdec: } & \forall P : \mathsf{Prop}. (P \lor \lnot P \rightarrow \lnot\lnot P) \\
\text{Lemma anddec: } & \forall P, Q : \mathsf{Prop}. (P \lor \lnot P) \rightarrow (Q \lor \lnot Q) \rightarrow \\
& (P \land Q) \lor \lnot (P \land Q) \\
\text{Lemma ordec: } & \forall P, Q : \mathsf{Prop}. (P \lor \lnot P) \rightarrow (Q \lor \lnot Q) \rightarrow \\
& (P \lor Q) \lor \lnot (P \lor Q) \\
\text{Lemma impdec: } & \forall P, Q : \mathsf{Prop}. (P \lor \lnot P) \rightarrow (Q \lor \lnot Q) \rightarrow \\
& (P \rightarrow Q) \lor \lnot (P \rightarrow Q).
\end{align*}

It is also preserved by bounded versions of the quantifiers. This means that proving decidability of predicates like Divides and Prime reduces to proving that the quantifiers in the defining terms can be bounded.

\begin{align*}
\text{Lemma alldec: } & \forall P : \mathbb{N} \rightarrow \mathsf{Prop}. \forall N : \mathbb{N}. (\forall n : \mathbb{N}. (P n) \lor \lnot (P n)) \rightarrow
\end{align*}
\[ \forall x : \mathbb{N}. (x < N \rightarrow (P_x)) \lor \neg \forall x : \mathbb{N}. (x < N \rightarrow (P_x)) \]

**Lemma **exdec: \( \forall P : \mathbb{N} \rightarrow \text{Prop}. \forall N : \mathbb{N}. (\forall n : \mathbb{N}. (P_n) \lor \neg (P_n)) \rightarrow \\
\exists x : \mathbb{N}. (x < N \land (P_x)) \lor \neg \exists x : \mathbb{N}. (x < N \land (P_x)). \]

Having developed the meta theory so far, it is possible to build the real mathematical theory needed for Pocklington’s criterion. The `divides`, `prime`, `mod`, `gcd`, `exp`, and `order` modules define the mathematical notions introduced in the definitions given above and contain many useful lemmata with proofs about these concepts. For instance, the main result in the module concerning prime numbers is the theorem stating that in order to prove primality of a natural number \( n \) it is enough to check divisibility by all the primes up to \( \sqrt{n} \):

**Lemma **primepropdiv: 
\[ \forall n : \mathbb{N}. (n > 1) \land (\forall p : \mathbb{N}. \text{Prime}(p) \land p | n \rightarrow (p > \sqrt{n})) \rightarrow \text{Prime}(n). \]

The `modprime` module contains some results about modulo arithmetic where the modulus is prime. The combination of the `modprime` and `order` modules could be replaced with additional effort by more abstract group theory modules. The `fermat` module contains Fermat’s little theorem.

Finally, the top module `pock` in Figure 1 contains the formal proof of Lemma 2.1. The proof proceeds using several technical lemmata which mimic the high-level reasoning of the informal proof. These technical lemmata are necessary for two reasons. First of all, the informal proof uses forward reasoning whereas the Coq system, being a goal-directed theorem prover, uses backward reasoning. In the backward reasoning style of theorem proving, the user is presented with a goal to prove and works his way back to the assumptions on which this goal depends by means of tactics. The informal proof, however, proceeds by contradiction introducing an arbitrary prime divisor \( p \) of \( n \). It shows that \( p > \sqrt{n} \), and from this it concludes that \( n \) must be prime. This forward style of reasoning can be simulated in Coq by first proving the lemma `primepropdiv`, cited above, and applying it to the current goal, \( \text{Prime}(n) \). Doing so, the goal is replaced by new obligations to prove that \( n > 1 \) and \( p > \sqrt{n} \) for all prime divisors \( p \) of \( n \). The application of the lemma corresponds to lines 1, 10, and 11 of the proof of Lemma 2.1.

The second reason for using technical lemmata in the construction of the proof of Pocklington’s criterion is to capture high-level reasoning. In fact, the steps in the informal proof are large steps. Look for example at line 4 of the informal proof. In order to show that \( q \) is the order of \( b \) in \( \mathbb{Z}_p^* \), a contradiction is derived from the assumption that \( b^q = 1 \mod p \) for some prime factor \( q_i \) of \( q \). This follows from a number of non-trivial technical lemmata such as:

**Lemma **order_ex: 
\[ \forall p : \mathbb{N}. \text{Prime}(p) \rightarrow \forall b : \mathbb{Z}. \exists d : \mathbb{N}. (\text{Order}(b, d, p)) \]

**Lemma **order_div: 
\[ \forall b : \mathbb{Z}. \forall x : \mathbb{N}. \forall p : \mathbb{N}. \text{Order}(b, x, p) \rightarrow \\
\forall y : \mathbb{N}. (y > 0 \land (b^y = 1 \mod p)) \rightarrow x | y \]

**Lemma **lemma3: 
\[ \forall a, b : \mathbb{N}. 0 < a < b \land a | b \rightarrow \exists q_i : \mathbb{N}. (q_i | b \land \text{Prime}(q_i) \land a | \frac{b}{q_i}). \]
The full formalization of Pocklington’s result gives us a way to generate proofs of primality that are formal and acceptable by the most skeptical approach.

3. Generating Primality Proofs

This section describes how Pocklington’s Criterion can be used to produce a formal and efficient primality proof for a relatively big\(^1\) prime number. In a skeptical approach one invokes an outside oracle to supply the theorem prover with the necessary witnesses for applying Pocklington’s criterion. It is natural to think of computer algebra systems acting as oracles when algebraic equalities have to be verified. For example, when \(a = b \pmod{n}\) needs to be proved the computer algebra system can provide \(q \in \mathbb{Z}\) such that \(a = b + qn\). For the skeptical approach to work, a computer algebra system must be able to supply both a fast decision for the primality of a positive number \(n\) and in the affirmative case the ability to provide additional extra information for building a proof-object.

To realize the automatic generation of primality proofs exploiting computer algebra systems, we implemented a Java applet which can communicate both with a computer algebra oracle and with Coq, see Figure 2 for an overview of the overall architecture. This implementation makes no claim of being a generic tactic creation framework. It was initiated as a case study to help build tool support for combining mathematical services provided by external engines, libraries and protocols such as (Göbel et al., 1999; Wang, 1999; Caprotti et al., 2000; JavaMath—Internet Accessible Mathematical Services, 2001; Solomon et al., 2001).

The applet allows the user to input a number \(n\) and generates (if \(n\) is prime) COQ tactics that show the primality of \(n\). The proof described by these tactics applies Pocklington’s criterion to the correct instantiation of the parameters. The parameters are witnesses retrieved from the computer algebra system. Communication with the chosen computer algebra system takes place using OpenMath (Dalmas et al., 1997; The OpenMath Consortium, 2000), a language for sharing mathematical objects between computer algebra systems, therefore any compliant computer algebra system can be used (Caprotti et al., 2000). The possibility of switching to a different engine is crucial since each computer algebra system is able to deal with arbitrary long integers under certain restrictions.

\(^1\)See the next section for a discussion on the size of the prime.
These restrictions vary, in particular the bounds on the probabilistic tests for primality and factorization are different. Depending on the chosen oracle, one is able to obtain witnesses for small (e.g. up 10 digits as in our Java implementation of the computer algebra oracle), big (e.g. up to 500 digits), or even bigger prime numbers (e.g. more than 500 digits). The generated tactics are sent to Coq and a formal proof-object is returned.

The applet can be seen as a prototype interactive mathematical document. The user can influence the presentation of the document in several ways. The prime number \( n \) can be set by the user, there are several computer algebra systems to choose from, and the user can specify to which extend the theorem prover should trust the witnesses provided by the computer algebra oracle. The latter choice is made by selecting a belief level ranging from skeptical (level 5) to believing (level 5). More precisely, we have selected the following classes of assertions within each belief level:

1. Believe nothing. All generated subgoals are proved.
2. Believe simple modular equations. As above, but modular equations of the form \( a \equiv b \pmod{n} \), that are proved directly by finding a witness, are assumed as axioms instead.
3. Believe modular equations. As above, but modular equations of the form \( a^m = b \pmod{n} \), that are proved using the divide and conquer method outlined below, are assumed as axioms instead.
4. Believe modular equations and linear combinations. As above, but also subgoals of the form \( \exists \alpha, \beta. (\alpha x + \beta x \equiv c \pmod{n}) \) are assumed as axioms.
5. Believe everything. Any subgoal may be assumed as an axiom.

In principle the proof-object returned by Coq could be parsed and presented using the tools developed in Oostdijk (2000) leading to a real interactive mathematical document in the sense of Caprotti and Oostdijk (2000).

Now we describe in detail the algorithm based on Pocklington’s criterion which reduces the proof of primality of a positive integer \( n \) to the proof of a number of algebraic identities. The PocklingtonC algorithm summarized in Figure 3 takes as input a prime number \( n \) and produces a Coq tactic script for constructing a proof-object for \( \text{Prime}(n) \). Interaction with computer algebra oracles takes place mostly in Step (1), where the witnesses are found, and (4), where the algebraic identities are shown.

When the computer algebra software is given a positive number \( n \), it first tests whether the number is indeed prime. If not, it returns the number. If the number \( n \) is prime, then the system can compute the numbers \( a, q \) and \( m \) as follows. For \( a \) take the primitive root \( \pmod{n} \), namely an element \( a \) such that \( a^{n-1} \equiv 1 \pmod{n} \) and \( a^i \not\equiv 1 \pmod{n} \) for \( i = 1, \ldots, n-2 \). For \( q \), consider the prime factorization \( n-1 = q_1 \ldots q_t \ldots q_k \), where \( q_1 \geq \cdots \geq q_t \geq \cdots \geq q_k \), and take \( q = q_1 \ldots q_t \) for the smallest \( t \) such that \( q \geq \sqrt{n} \). Finally, for \( m \) take \( m = (n-1)/q \). All the operations to compute the appropriate \( a, q = q_1 \ldots q_t \), and \( m \) are carried out by the computer algebra package upon receiving the prime number \( n \). Notice that these witnesses, computed as described, satisfy the hypotheses of Pocklington’s criterion. Subgoals (4)(a) and (4)(c) are clearly true. Condition (4)(b) is true because \( n \) is prime and \( \gcd(a^{n-1}/q_i - 1, n) \) cannot be \( n \). If it was \( n \), then \( a^{n-1}/q_i \equiv 1 \pmod{n} \) for an exponent \( n-1 \). However, this is not possible because \( a \) is the primitive root \( \pmod{n} \).

The computer algebra oracle is also called in Step (4)(b). It computes the coefficients for the linear combinations generated by the gcd proof obligations using a straightforward
Figure 3. Pocklington criterion algorithm.

extension of the Euclidean gcd algorithm. Most computer algebra systems provide this
algorithm as primitive. The oracle also computes the result of the exponentiation
for a \( n \) and for all the intermediate steps in the divide and conquer
procedure outlined below. Intermediate obligations are of the form \( x \equiv y \pmod{n} \)
where all variables are concrete instances such that \( x, y \leq n \). Although finding the witness
\( z \in \mathbb{Z} \) such that
\[
x^m \equiv y + z \cdot n \pmod{n}
\]
is easy for the computer algebra system, the computations
involved in proving the equality directly are too expensive for Coq as \( x^m \) becomes large.
Instead, the goal is changed by replacing the exponent \( m \) as follows.

\[
x^m \equiv y \pmod{n} \iff \begin{cases}
x^z = z \pmod{n}, \quad zz = y \pmod{n} & \text{if } m \text{ even} \\
x^{m-1} = z \pmod{n}, \quad xzz = y \pmod{n} & \text{if } m \text{ odd}
\end{cases}
\]
The computer algebra oracle is used to compute \( z \) such that \( 0 \leq z < n \). The resulting goal
involving \( x \) is solved by recursively applying this procedure, the other goal can be proved
directly as all numbers are small. Note that this solution again relies on the computer
algebra oracle to find witnesses.

To summarize the overall picture, the algorithm \textit{PocklingtonC} can be used to produce
a Coq tactic script that generates a proof-object for the primality of a positive number.
The only requirements on the computer algebra systems used as oracles is the ability
to perform integer computations like prime testing, factorization, gcd computation
and some modular arithmetic. Since the communication uses the \textit{OpenMath} standard, the
architecture allows for multiple computer algebra oracles, see Figure 2. The tactics view of the applet presents the generated tactic script to the user, see Figure 4.
Table 1. Easy and hard primes.

<table>
<thead>
<tr>
<th>n</th>
<th>Recursive calls</th>
<th>Believing</th>
<th>Skeptical</th>
</tr>
</thead>
<tbody>
<tr>
<td>2939</td>
<td>3, 7, 113</td>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>4111</td>
<td>17, 137</td>
<td>4</td>
<td>25</td>
</tr>
<tr>
<td>7829</td>
<td>17, 103</td>
<td>4</td>
<td>28</td>
</tr>
<tr>
<td>4079</td>
<td>3, 7, 127, 509, 1019</td>
<td>9</td>
<td>61</td>
</tr>
<tr>
<td>7727</td>
<td>3, 103, 1031, 3863</td>
<td>7</td>
<td>52</td>
</tr>
</tbody>
</table>

All responses of Coq can be predicted, so the tactic script can be composed without consulting Coq. Once the script is generated, the user can send it to Coq which returns a proof-object. The proof-object is presented in the proof-object view, see Figure 5.

4. Results

Our implementation of the architecture described above consists of a Java application in which the user enters a positive integer \( n \) and selects a computer algebra package running on a remote server. If the number is prime, the computer algebra package is repeatedly invoked for a concrete value of \( n \) and for the subsequent recursive calls of the factors. The application then generates a Coq tactic script that can be sent automatically to Coq when proving the goal \( \text{Prime}(n) \).

In practice, the algorithm outlined in Section 3 has to take into account limitations on the size of the prime number \( n \). Computer algebra software, like GAP, is able to test primality for integers that are up to 13 digits long. For bigger integers, the primality test are probabilistic and return a probable prime. For instance, testing numbers with several hundreds digits is quite feasible in GAP 4 using IsPrimeInt or IsProbablyPrimeInt. Concerning factorization, FactorsInt is guaranteed to find all factors less than \( 10^{10} \) and will find most factors less than \( 10^{10} \).

We tested the generated tactic scripts for all primes between 2 and 7927 (the first 1000 primes) and measured the time needed by Coq to run the tactic script and check the resulting proof-object on a Unix workstation. Obviously the general trend is that larger primes need more time. However, some numbers are much harder due to an unfortunate prime factorization of \( q \). Some examples of easy and hard primes are given in column 1 in Table 1 followed by the number of recursive calls, and the seconds needed to prove primality using a believing and a skeptical approach.

For example when proving Prime(2039), the algorithm is forced to choose \( q = 1019 \), since the prime factorization of 2038 is \( 2 \cdot 1019 \). Now, 1018 in its turn has as prime factorization \( 2 \cdot 509 \). In the end recursive calls for 3, 7, 127, 509, and 1019 are needed to prove Prime(2039), and it takes about 45 seconds to verify the proof. In contrast to verify the proof generated for \( n = 2939 \) only needs recursive calls for 3, 7, and 113 and only takes about 25 seconds. The number of primes required in the recursion steps for the first 1000 primes are plotted in Figure 6. Numbers of the form \( 2^n + 1 \) require no recursive call (they are 3, 5, 17, 257, ...). Next best are primes of the form \( p + 1 \)

\(^\dagger\)It remains to see whether, for such titanic primes, Coq succeeds in checking the generated proofs.

\(^\ddagger\)The machine used for the tests is a Sun Ultra 10 with 333 Mhz Sparc processor and 128 MB memory. The Coq version used is 6.3.1.
where the factorization of $p$ involves primes of the form $2^n + 1$ above (they are 7, 11, 13, 19, 37, 41, ...), and so on. In general, given a number $n$, the worst case is that there will be $\lceil \log_2 \frac{n+1}{2} \rceil$ primes to be checked recursively. This trend shows also in the plot of the size of the context used in the primality proof (number of lemmata about modular equations, linear combinations, and primes) as given in Figure 7. The complete run time results for the first 1000 primes are plotted in Figure 8. The timings (in seconds) for the believing approach (belief level 4 on p. 62, where subgoals to prove modular equations and linear combinations are not proved) and for the skeptical approach (belief level 1 on
p. 62, where all subgoals are proved) are given. The results in Figure 8 show that the current implementation significantly improves the implementation discussed in Caprotti and Oostdijk (2000). This is due to the fact that the current implementation avoids the unary represented numbers completely, whereas the generated tactic script in Caprotti and Oostdijk (2000) still used unary represented numbers in some places. In Caprotti and Oostdijk (2000) the time complexity of the algorithm seemed to be worse than linear, while Figure 8 shows a logarithmic time complexity.

Some large primes that were tested are listed in Table 2. For several of them, GAP issues a warning that a probabilistic primality test is used. The last columns lists the time Coq needed to check the proof-object produced by the skeptical approach.

5. Conclusions

We have shown how by combining computer algebra oracles and theorem provers it is possible to automatically produce proofs of primality that are efficiently and formally verifiable. The primality proofs are obtained according to Pocklington’s criterion that has been formally proven in Coq. The formal development has been carried out constructively and is now available as contributed library of Coq, it consists of 260 smaller lemmata taking approximately 5000 lines of code.
The architecture for using Pocklington’s criterion relies on the use of computer algebra oracles. We implemented these oracles by reusing OpenMath mathematical servers providing computational capabilities on the network that we have developed previously. The JAVA application that produces the tactic script behaves as a client to these servers. In this general view, our experiments are an example of how to use computer algebra in theorem proving and an investigation on the tools that are required to effectively carry out the integration. We profited greatly from our former work in using the standard communication language OpenMath to interface to a variety of symbolic computation systems. For some of the systems we used, we have developed our own small OpenMath interface, available from http://crystal.win.tue.nl/public/projects, since unfortunately most systems do not yet support OpenMath (GAP is among the few that actually do). It was extremely easy to extend the already available OpenMath servers to handle the requests of the Pocklington’s applet. In our experience, adding a new oracle for the specific tasks of this case study is quite straightforward: the few OpenMath objects used in the communication are the only ones that need to be translated, together with the result. It should eventually be possible in future to interact with mathematical servers by a unique protocol without having to worry about syntax, location, or availability.

We tested the current implementation with primes up to 44 digits. Besides the difficulty of computing the witnesses required by the proof of such large numbers, another issue has been the time required to check larger proof-objects. It is striking to see what can actually be accomplished by combining powerful computer algebra systems with Coq. In fact, we do not think that Ferrier’s prime is the largest prime that can be tackled by our approach although we have not yet attempted any bigger one. We are currently adding interfaces to Pari and to Magma which will allow to tackle larger problems†. For this application, adding computer algebra engines as oracles is considerably simpler than changing the proof assistant, which would imply doing most of the formal development plus designing a new ad hoc tactic generator. It is our hope that case studies such as this one stimulate the developers of computational software in building systems that provide mathematical services in a standardized way. This would greatly simplify this kind of investigation.

Acknowledgements

The authors are very grateful to Henk Barendregt and to Arjeh Cohen for suggesting this case study and for challenging and inspiring them with proving bigger and bigger primes. Herman Geuvers has helped us with hints during many discussions. A grateful thanks to the anonymous referees for pointing out missing references in the initial version of the bibliography.

References


†For instance the primitive root of the prime 3141592653589793238462643383279528841 is computed without problems by Magma, whereas GAP complains.


*Originally Received 5 November 2000*  
*Accepted 15 February 2001*