Proving Pearl: Knuth's Algorithm for Prime Numbers

Laurent Théry

INRIA, 2004 route des Lucioles, 06902 Sophia Antipolis France*
`Laurent.Thery@sophia.inria.fr`

Abstract. In his book "The Art of Computer Programming", Donald Knuth gives an algorithm to compute the first n prime numbers. Surprisingly, proving the correctness of this simple algorithm from basic principles is far from being obvious and requires a wide range of verification techniques. In this paper, we explain how the verification has been mechanised in the Coq proof system.

1 Introduction

There is no relation between the length of a program and the difficulty of its proof of correctness. Very long programs performing elementary tasks could be trivial to prove correct, while short programs relying on some very deep properties could be much harder. Highly optimized programs usually belong to the second category. For example, algorithms designed for efficient arithmetic operations are known to be hard to verify since every single line has been thought in order to minimize execution time and/or memory allocation. Bertot et al. [4] illustrate the difficulty of verifying such algorithms.

In this paper we are interested in an algorithm given by Knuth in his book "The Art of Computer Programming" [12]. This algorithm takes an integer n as an argument and returns the list of the first n prime numbers. The correctness of this simple algorithm relies on a deep property of prime numbers called Bertrand's postulate. The property, first conjectured by Bertrand and then proved by Chebyshev, states that for any integer number $n \geq 2$ there always exists a prime number p strictly between n and $2n$. Proving Knuth's algorithm from basic principles means formally proving Bertrand's postulate. To do so we follow the proof given by Arkadii Slinko [15]. The original idea of this elementary proof is due to Paul Erdös [6]. The proof itself has a very interesting structure. The initial problem in number theory is translated into real analysis, namely analysing the variation of a function. Using derivative and the intermediate value theorem, it is possible to conclude that for n greater than 128 the property holds. To finish the proof, we are then left with the task of individually checking that the property holds for n varying from 2 to 127.

The paper is structured as follows. In Section 2, we show how prime numbers can be easily defined in a prover. In Section 3, we present the algorithm proposed

* Currently Visiting Professor at University of L'Aquila, Italy.

by Knuth. In Section 4, we detail the different logical assertions that need to be attached to the program to prove its correctness. In Section 5, we outline the proof of Bertrand's postulate. In Section 6, we comment on some specific aspects of our formalisation.

2 Prime Numbers

The notion of primality can be defined in a prover in a simple way. Natural numbers are usually defined using Peano representation. In Coq, we have:

Inductive $\mathbb{N} : Set :=$
  $O : \mathbb{N}$
| $S : \mathbb{N} \to \mathbb{N}.$

With this definition, 0,1,2 are represented as O, $(S\ O)$, $(S\ (S\ O))$. The next step is to define the notion of divisibility:

Definition $divides\colon \mathbb{N} \to \mathbb{N} \to Prop := \lambda a, b\colon \mathbb{N}.\, \exists q\colon \mathbb{N}.\, b = qa.$

In Coq, predicates are represented as functions. The predicate *divides* is a function that takes two natural numbers and returns a proposition. The fact that *a divides b* is then written $divides(a, b)$ and corresponds to the proposition $\exists q\colon \mathbb{N}.\, b = qa.$

Once divisibility has been defined, we can proceed with primality. A number is prime if it has exactly two divisors 1 and itself:

Definition $prime\colon \mathbb{N} \to Prop := \lambda a\colon \mathbb{N}.$
  $a \neq 1 \wedge (\forall b\colon \mathbb{N}.\, divides(b, a) \Rightarrow (b = 1 \vee b = a)).$

With this definition it is possible to derive some basic properties of prime numbers. Two of them are of special interest in our context. The first one states that all prime numbers are odd except 2. Using the definition of odd number:

Definition $odd\colon \mathbb{N} \to Prop := \lambda a\colon \mathbb{N}.\, \exists b\colon \mathbb{N}.\, a = 2b + 1.$

we have the following theorem:

Theorem $prime2Odd\colon \forall p\colon \mathbb{N}.\, prime(p) \Rightarrow p = 2 \vee odd(p).$

The second property states that a number n is prime if all the prime numbers less than $\sqrt{n}$ do not divide it:

Theorem $primeDef_1\colon \forall n\colon \mathbb{N}.$
  $1 < n \wedge (\forall p\colon \mathbb{N}.\, prime(p) \wedge p^2 \leq n \Rightarrow \neg(divides(p, n))) \Rightarrow prime(n).$

The bound of $\sqrt{n}$ comes from the fact that if n is composite, i.e. $n = pq$, then either p or q must be less than $\sqrt{n}$.

3 Knuth's Algorithm

To express the algorithm given by Knuth and state its correctness, we use the Why tool [7]. This tool takes a program annotated with logical assertions à

```
parameter n: int
parameter a: array n of int
parameter m,s,i,j:  int ref
parameter b: bool ref

external sqr : int -> int
external mod : int -> int -> int

begin
  a[0] := 2;
  m := 3;
  i := 1;
  while ((!i) < n) do
    b := true;
    s := (sqr !m);
    j := 0;
    while ((!b) && a[!j] <= !s) do
      if (mod !m a[!j]) = 0
      then b := false
      else j := !j + 1
    done;
    if (!b) then
      begin
        a[!i] := !m;
        i := !i + 1
      end;
    m := !m + 2
  done
end
```

Fig. 1. The Algorithm in Why

la Hoare [10] and generates a list of verification conditions. Proving all these conditions ensures that all the logical assertions in the program hold. Why is generic in the sense that it is not linked to a specific prover. Outputs for Pvs and Coq are available.

The algorithm written in Why is given in Figure 1. The syntax of Why is a subset of the one of the Ocaml programming language. In Ocaml, a variable that is modifiable has a reference type α *ref*. If `x` is a variable of type *int ref*, the expression `!x` denotes the value of `x` and the statement `x := !x +1` increments the value of `x` by one. Let us explain the program given in Figure 1. It starts with a sequence of declarations:

```
parameter n: int
parameter a: array n of int
parameter m,s,i,j:  int ref
parameter b: bool ref
```

According to these declarations, `n` is a variable whose value cannot be modified, `a` is an array that should eventually contain the first `n` primes, `m`, `s`, `i` and j are modifiable integer variables, and `b` is a modifiable boolean variable. To write the algorithm, we need two extra functions on integer numbers. Since these functions are not going to be defined, they are declared external:

```
external sqr : int -> int
external mod : int -> int -> int
```

The first one represents the square root, the second one the modulo. For example `(sqr 5)` and `(mod 23 7)` are both equal to 2.

The program has two `while` loops. The outer one fills the array `a` with prime numbers. For this, it uses a candidate prime number `m`. The boolean variable `b` tells whether `m` is prime or not. If at the end of the inner loop, the value of `b` is true, `m` is put in the array. In any case at the end of each iteration of the outer loop, the value of `m` is incremented by 2.

The inner loop checks the primality of the value of `m`. For this it makes use of the property *primeDef*$_1$. The test `(mod !m a[!j])=0` is used for checking if `a[j]` divides `m`. The real difficulty in proving the correctness of the program lies in the following line:

```
      while ((!b) && a[!j] <= !s) do
```

If the guard of the loop had been more defensive, i.e.

```
      while ((!b) && j<i && a[!j] <= !s) do
```

the correctness of the program would be a direct consequence of the two properties *prime2Odd* and *primeDef*$_1$. As noted by Knuth the test `j<i` is unnecessary because of the density of prime numbers. To find a new prime number for the location `a[i]`, the program starts from the value `a[i-1]+2` incrementing `m` repeatedly by 2 till a prime number is found. It results that `j` could exceed `i` if and only if there was no prime number between `a[i-1]` and `a[i-1]`^{2}. Bertrand's postulate ensures that there is always a prime between `a[i-1]` and 2`a[i-1]`. As `a[i-1]` is prime and thus larger than 1, we have 2`a[i-1]` $\leq$ `a[i-1]`^{2}, so `j` cannot exceed `i`.

4 Correctness

In order to prove the program given in Figure 1, we have to annotate it with logical assertions. To do so we need some predicates and functions:

```
logic one : bool -> int
logic In : int,array int,int,int -> prop
logic Prime : int -> prop
logic Odd : int -> prop
logic Divides : int,int -> prop
```

The function `one` transforms a boolean into an integer: *true* $\rightarrow 1$ and *false* $\rightarrow 0$. It is used to express some termination properties of the program.
The predicate `In` is used to express properties of the array: `In(n,a,i,j)` is equivalent to the fact that there exists an index $\mathtt{i} \leq \mathtt{k} < \mathtt{j}$ such that $\mathtt{a[k]} = \mathtt{n}$. The predicates `Prime`, `Odd` and `Divides` are the usual predicates on integers.

4.1 Preconditions and Postconditions

The only precondition for the program is

```
      { 0<n }
```

It is needed to ensure the correct execution of the statement `a[0]=2`. The postcondition simply states that at the end of the program `a` should hold a complete ordered list of prime numbers:

```
{(forall k:int.  (0 <= k and k < n -> Prime(a[k])))       and
 (forall k:int. forall j:int.
       (0 <= k and k < j and j < n -> a[k] < a[j]))       and
 (forall k:int.
       (0 <= k and k <= a[n-1] and Prime(k)) -> In(k,a,0,n))
}
```

4.2 Invariant and Variant for the First Loop

For the first loop

```
      while ((!i) < n) do
```

the invariant is a conjunction of three blocks. The first one states that the final assertion holds till `i`:

```
(forall k:int.  (0 <= k and k < i -> Prime(a[k])))        and
(forall k:int. forall j:int.
       (0 <= k and k < j and j < i -> a[k] < a[j]))       and
(forall k:int.
       (0 <= k and k <= a[i-1] and Prime(k)) -> In(k,a,0,i))
```

The second block keeps the information related to `m`, i.e. `m` is odd and in the interval defined by Bertrand's postulate:

```
      a[i-1] < m and m < 2*a[i-1] and Odd (m)
```

The last block keeps the information related to `i`:

```
      0 < i and i <= n
```

In order to find the variant that ensures the termination of the first loop, we have to notice that either `i` gets closer to `n` or `m` gets closer to 2`a[i-1]`. So for a lexicographic order, the pair of these two quantities always gets smaller. This gives us the following variant:

```
      variant (n-i, 2*a[i-1]-m) for lexZ
```

4.3 Invariant and Variant for the Second Loop

For the second loop

```
while ((!b) && a[!j] <= !s) do
```

the invariant is a conjunction of two blocks. The first block keeps the information related to `j`:

```
0 <= j and j < i
```

The second block states that if the boolean `b` is true we have not yet found a divisor of `m` and if it is false, `a[j]` divides `m`:

```
(if (b)
   then (forall k:int.
            (0 <= k and k < j -> not(Divides(a[k],m))))
   else  Divides(a[j],m))
```

The termination is ensured because `j` always gets closer to `i` except when `b` is set to false:

```
variant one(b)+i-j
```

This ends all the assertions we need to put in the program. The complete annotated program is given in Appendix A.

5 Bertrand's Postulate

Running the Why tool on the annotated program given in Appendix A generates 18 verification conditions. Only the condition coming from the invariant of the first loop that says that `m` keeps between `a[i-1]` and `2a[i-1]` is difficult to prove. This is no surprise: to prove it we need a proof of Bertrand's postulate.

In this section we are not going to present the whole formal proof but only illustrate some of its most interesting aspects. The proof is largely inspired from that given by Arkadii Slinko [15]. Still, it has been slightly modified so to make its formalisation in a prover easier. We refer to the technical report [16] for the details of the proof.

5.1 Number Theory

The main part of the proof consists in proving properties about natural numbers. Primality is a key notion on which we need to be able to compute. For this we turn the predicate into a test function. The primality test $1_p(a)$ is defined as follows:

$1_p(a) = a$ if a is prime
$1_p(a) = 1$ otherwise

In particular this test is used to express theorems about the product of prime numbers. For example, to denote the product of all prime numbers less than n we simply write

$$\prod_{i \leq n} 1_p(i)$$

Another important notion in the proof of Bertrand's postulate is the one of binomial coefficients. The main step of the proof is to find an upper bound and a lower bound for $\binom{2n}{n}$. The lower bound is given by the following theorem:

Theorem *binomialEven*: $\forall n\colon \mathbb{N}.\, 0 < n \Rightarrow 4^n \leq 2n\binom{2n}{n}$.

The upper bound is only valid in a context where there is no prime between n and $2n$ and where n is greater than 128:

Theorem *upperBound*:
$\forall n\colon \mathbb{N}.\, 2^7 \leq n \wedge (\forall p\colon \mathbb{N}.\, n < p < 2n \Rightarrow \neg(prime(p))) \Rightarrow$
$\binom{2n}{n} \leq (2n)^{\sqrt{2n}/2-1} 4^{2n/3}$.

In order to define binomial coefficients, we use Pascal's triangle:

$$\begin{aligned} \binom{0}{a} &= 0 \qquad \text{if } a \neq 0 \\ \binom{a}{0} &= 1 \\ \binom{a+1}{b+1} &= \binom{a}{b+1} + \binom{a}{b} \end{aligned}$$

We then derive the usual closed form:

Theorem *binomialFact*: $\forall n, m\colon \mathbb{N}.\, \binom{n+m}{m} n! m! = (n+m)!$.

Three main properties of binomial coefficients are needed:

Theorem *binomialMonoS*: $\forall n, m, p\colon \mathbb{N}.\, 2m < n \Rightarrow \binom{n}{m} \leq \binom{n}{m+1}$.

Theorem *binomialComp*: $\forall n, m\colon \mathbb{N}.\, \binom{n+m}{n} = \binom{n+m}{m}$.

Theorem *primeDiracDividesBinomial*:
$\forall n, m, p\colon \mathbb{N}.\, n < p \leq n + m \wedge m < p \Rightarrow divides(1_p(p), \binom{n+m}{n})$.

The first property is a direct consequence of the definition while the second property follows from the closed form. To get the third property we use the closed form and the fact that, if a prime number divides a product, then it divides one of the elements of the product.

The road to the lower bound theorem *binomialEven* is rather direct. It starts by proving the binomial theorem:

Theorem *expPascal*: $\forall a, b, n\colon \mathbb{N}.\, (a+b)^n = \sum_{0 \leq i \leq n} \binom{n}{i} a^i b^{n-i}$.

By applying this theorem for $a = b = 1$ we get

Theorem *binomial*$_2$: $\forall n\colon \mathbb{N}.\, 2^n = \sum_{0 \leq i \leq n} \binom{n}{i}$.

Now we have all the ingredients to prove the theorem *binomialEven*. Using the theorem *binomial*$_2$, we have

$$2^{2n} = \sum_{0\le i\le 2n} \binom{2n}{i}$$

Taking apart the first and the last term gives

$$2^{2n} = 1 + \sum_{1\le i\le 2n-1} \binom{2n}{i} + 1$$

We have $2 \le \binom{2n}{n}$ and using *binomialMonoS* and *binomialComp* we can prove that

$$\binom{2n}{i} \le \binom{2n}{n} \text{ for } 1 \le i \le 2n-1$$

Using these two bounds we have

$$4^n \le \binom{2n}{n} + (2n-1)\binom{2n}{n} = 2n\binom{2n}{n}$$

The proof of the upper bound theorem *upperBound* is much more intricate. In this paper we only illustrate how the expression $4^{2n/3}$ is derived. It comes from an appropriate upper bound on the product of prime numbers less than $2n/3$. In order to establish this upper bound, we first prove a dual theorem to the theorem *binomialEven* and get

Theorem *binomialOdd*: $\forall n\colon \mathbb{N}.\ \binom{2n+1}{n} \le 4^n$.

The proof follows the same path as the one of *binomialEven*. We first use the theorem *binomial*$_2$:

$$2^{2n+1} = \sum_{0\le i\le 2n+1} \binom{2n+1}{i}$$

By keeping only the two middle terms of the sum, we have

$$\binom{2n+1}{n} + \binom{2n+1}{n+1} \le 2^{2n+1}$$

Using the theorem *binomialComp*, we get

$$2\binom{2n+1}{n} \le 2^{2n+1}$$

By simplifying by 2 on both sides we get the expected result.

Now it is possible to establish the expected upper bound on the product of prime numbers:

Theorem *prodPrimeLt*: $\forall n\colon \mathbb{N}.\ 1 < n \Rightarrow \prod_{i\le n} 1_p(i) < 4^n$.

The proof is done using complete induction on $<$. First of all, the property is true for $n = 2$. We suppose that the property holds for all $m < n$ and try to prove that the property holds for n.
If n is even, we have

$$\prod_{i\le n} 1_p(p) = \prod_{i\le n-1} 1_p(p) \le 4^{n-1} \le 4^n$$

If n is odd, we can write n as $2m+1$ and we have

$$\prod_{i\leq 2m+1} 1_p(p) = (\prod_{i\leq m+1} 1_p(p)) \quad (\prod_{m+2\leq i\leq 2m+1} 1_p(p))$$

Using the induction hypothesis on the left element of the product, we get

$$\prod_{i\leq 2m+1} 1_p(p) \leq 4^{m+1} \quad (\prod_{m+2\leq i\leq 2m+1} 1_p(p))$$

From the theorem *primeDiracDividesBinomial* we know that each element in the product $\prod_{m+2\leq i\leq 2m+1} 1_p(p)$ divides $\binom{2m+1}{m}$. So we have

$$\prod_{i\leq 2m+1} 1_p(p) \leq 4^{m+1}\binom{2m+1}{m}$$

It is sufficient to apply the theorem *binomialOdd* to get the expected result

$$\prod_{i\leq n} 1_p(p) \leq 4^{m+1}4^{m} = 4^{2m+1} = 4^{n}$$

5.2 Real Analysis

The two theorems *binomialEven* and *upperBound* give a lower bound and an upper bound for $2n\binom{2n}{n}$ respectively. By composing them, we get the following inequality:

$$4^n \leq 2n\binom{2n}{n} < (2n)^{\sqrt{2n}/2}4^{2n/3}$$

For n sufficiently large, this inequality cannot hold since elementary function analysis tells us that the left part of the inequality, 4^n, grows much faster than the right part, $(2n)^{\sqrt{2n}/2}4^{2n/3}$. To prove this formally, we transfer the problem into real analysis. So far all our definitions and theorems have been using natural numbers only. For example, in the previous inequality the operations $\sqrt{\ }$ and $/$ are functions over natural numbers, i.e $\sqrt{6} = 2$ and $7/2 = 3$. Now if we are able to prove that for x real and $x \geq 128$ we have

$$(2x)^{\sqrt{2x}/2}4^{2x/3} < 4^x$$

where the operations $\sqrt{\ }$ and $/$ are the usual real functions then, because of monotonicity, the inequality over the natural numbers should also hold. This means that the only way the theorem *upperBound* could still be true is because the assumption that there is no prime number between n and $2n$ is false. So Bertrand's postulate would be true for $n \geq 128$.

To prove the inequality over the reals, we first simplify it using properties of the power function

$$(2x)^{\sqrt{2x}/2} < 4^{x/3}$$

By taking the logarithm on both sides, we have

$$\frac{\sqrt{2x}}{2}\ln(2x) < \frac{2x}{3}\ln(2)$$

By simplifying it again we get

$$0 < \sqrt{8x}\ln(2) - 3\ln(2x)$$

If we set $f(x) = \sqrt{8x}\ln(2) - 3\ln(2x)$, we are left with proving that this function is always strictly positive for $128 \leq x$. If we evaluate $f(128)$, we get

$$\begin{aligned} f(128) &= f(2^7) \\ &= \sqrt{2^{10}}\ln(2) - 3\ln(2^8) \\ &= 2^5\ln(2) - 3\,2^3\ln(2) \\ &= 2^3\ln(2)(4-3) > 0 \end{aligned}$$

If we compute the derivative of f we get

$$f'(x) = \frac{8\ln(2)}{2\sqrt{8x}} - 3\frac{2}{2x} = \frac{\sqrt{2x}\,\ln(2) - 3}{x}$$

It is easy to show that the derivative is positive for $128 \leq x$. This implies that the function is positive. The above function analysis demonstrates that Bertrand's conjecture holds for $128 \leq n$. Checking individually the cases from 2 to 127 gives us the main result:

Theorem *Bertrand*: $\forall n \colon \mathbb{N}.\, 2 \leq n \Rightarrow \exists p \colon \mathbb{N}.\, \mathit{prime}(p) \wedge n < p < 2n.$

6 Some Remarks on the Formal Development

The full development is available at
`ftp://ftp-sop.inria.fr/lemme/Laurent.Thery/Bertrand/index.html`
It is 6000 lines long. The actual proof of the 18 verification conditions is 1000 lines long of which 800 are just the statements of the conditions. The development largely benefits from a previous formalisation of the correctness of the RSA encryption algorithm [3] in which binomials were defined in order to prove Fermat's little theorem. From this previous development 1000 lines were reused.

The really difficult part of the formalisation is the proof of Bertrand's postulate. With respect to the proof on paper [15], we discover that the applicability of two properties should be restricted. The theorem *powerDivBinomial*$_3$ was clearly not valid for $n = 1$. More interesting was the case of the theorem *upperBound*. In the paper the condition $128 \leq n$ was missing. The condition that $n \geq 128$ was only introduced later in the function analysis. In the proof of the theorem *upperBound* it was wrongly stated that the number of prime numbers less than p was always less than $p/2 - 1$. Spotting such minutiae is a clear benefit of formalising proofs with the help of a proof assistant. It shows that, when possible, a mechanised proof is a valuable companion to a proof on paper.

Formalising the analysis of the function f requires basic notions about the exponential and logarithmic functions. It also needs the fact that a function that has a positive derivative is increasing. This is a consequence of the intermediate value theorem. The most tedious part of the function analysis was actually to prove that the inequality that we had over the natural numbers should also hold for the real numbers. In Coq, natural numbers and real numbers being two separate types, translating statements from natural to real numbers must be justified explicitly.

When trying to prove the property that there is always a prime number between n and $2n$ for $n \leq 128$, we took full advantage of the possibility of defining functions that can be directly evaluated inside Coq. For example, we have defined the function *primeb* of type $\mathbb{N} \rightarrow$ *bool* and proved its associated theorem of correctness

Theorem *primebCorrect*: $\forall n$: *nat*.
  if $primeb(n)$ **then** $prime(p)$ **else** $\neg prime(p)$.

This theorem says that if $primeb(n)$ evaluates to *true* we have a proof of $prime(p)$ and if $primeb(n)$ evaluates to *false* we have a proof of $\neg prime(p)$. Coq is based on the isomorphism of Curry-Howard. This means that proofs are programs. So the proof of the theorem *primebCorrect* is actually a program that, given a natural number n, returns a proof of primality or a proof of non-primality. It means, for example, that since *primeb*(*11*) evaluates to *true* a proof of $prime(11)$ is simply the proof term *primebCorrect*(*11*), i.e. the application of the program *primebCorrect* to the argument *11*. Using functions that can be evaluated inside Coq is interesting because not only it automates proofs but also generates very small proof objects. Following the same idea we have a function *checkPostulate* with the associated theorem of correctness

Theorem *checkPostulateCorrect*: $\forall m$: $\mathbb{N}$.
  $checkPostulate(m) = true \Rightarrow$
    $\forall n\colon \mathbb{N}.\, 2 \leq n \leq m \Rightarrow \exists p\colon \mathbb{N}.\, prime(p) \wedge n < p < 2n.$

Then to prove the following theorem

Theorem *postulateCorrect*128:
  $\forall n\colon \mathbb{N}.\, 2 \leq n \leq 128 \Rightarrow \exists p\colon \mathbb{N}.\, prime(p) \wedge n < p < 2n.$

we use the previous theorem with a proof of $checkPostulate(128) = true$. Since the function *checkPostulate* can be evaluated directly inside Coq, the expression $checkPostulate(128) = true$ is identical to the expression $true = true$. Its proof is an instantiation of the theorem *reflEqual* that states the reflexivity of equality. The proof term for the theorem *postulateCorrect128* is then *postulateCorrect*(128, *reflEqual*(*bool*, *true*)).

The formal development also includes the proof of a surprising corollary of Bertrand's postulate suggested to us by Gérard Huet. For any given n, when taking the set of all natural numbers from 1 to $2n$, it is always possible to sort them pairwise in such a way that the sum of each pair is a prime number. For example, with $n = 10$ we have

$$\begin{array}{c} \{ \quad (1,2); \quad (3,4); \quad (5,8); \quad (6,7); \; (9,14); \\ (10,13); \, (11,12); \, (15,16); \, (17,20); \, (18,19) \; \} \end{array}$$

The proof of this corollary is left to the reader and can be found in the file `Partition.v` of our formal development.

7 Conclusions

We believe that Knuth's algorithm is a very nice example of both the complexity and the diversity of program verification. First of all, it is elementary. Prime numbers are among the first examples one encounters when learning programming. They are also a standard example for theorem proving. From the point of view of program verification, Knuth's algorithm shows that adding or removing a single instruction in a program can have a huge impact on the difficulty of establishing its correctness. With the extra test in the second loop, proving Knuth's algorithm correct would be a relatively easy exercise. Note that even if Bertrand's postulate were already in the database of the theorem prover, deriving the correctness of Knuth's algorithm automatically would still require some non-trivial insight. Bertrand's postulate asserts that there is a prime between n and $2n$. In the program what is needed is that there exists a prime between n and n^2.

From the point of view of theorem proving, the interesting part of Knuth's algorithm is the proof of Bertrand's postulate. In order to prove it, we had to derive a fair amount of properties in discrete mathematics. This was no surprise. More surprisingly the final steps required to transfer the problem into real analysis. As we could only assess that the property holds asymptotically, we had to establish that the property holds for an initial segment of the integers. Luckily enough we had only to check the property from 2 to 128. This property has been proved automatically using the evaluation mechanism of functions inside Coq. It is worth noticing that slightly different versions of the proof exist that give a different initial segment to check. For example in the book "Proofs from THE BOOK" [2] the bound is $4000 \leq n$. The function we have written to solve the problem for $n \leq 128$ is very naïve and uses a brute force method. It would be too inefficient for $n \leq 4000$. A systematic and more efficient method proposed by Harrison and Théry [9] would be to delegate the search for all the necessary primes to an external program and use Coq only to check the result. Note that a similar method [5] has already been used to get relatively large prime numbers in Coq.

The main contribution of this work is to present a tour of different formal verification techniques with an elementary example, a proving pearl. The main difficulty of proving Knuth's algorithm correct lies in having all these techniques working together. A much more elaborate proof effort that exhibits a similar diversity of verification techniques is the one described by John Harrison [8]. The fact that it is possible to carry out such kinds of verification in a single system shows how generic provers based on higher-order logics are. Having expressive logics makes them suitable for all different kinds of verification. It also indicates the maturity of theorem provers.

Number theory is an area of interest for theorem proving. For example, Art Quaife [14] shows how it is possible to prove automatically some non-trivial theorems of number theory. More recently, Joe Hurd [11] gives a very nice formalisation of the correctness of Miller-Rabin probabilistic algorithm to check primality. Along this line, the next natural candidate for formal verification is the recent polynomial algorithm to check primality proposed by Manindra Agrawal, Neeraj Kayal and Nitin Saxena [1]. Unfortunately their algorithm relies on much deeper properties than Bertrand's postulate, such as Brun-Titchmarsh theorem. These properties require the formalisation of some fascinating but elaborate tools, such as sieve methods. It is then most likely that mechanising the correctness of such an algorithm will still be a challenge for the next couple of years.

Acknowledgments. The formalisation in this paper was motivated by the preparation of an introductive lecture on Formalised Mathematics that the author gave at the Types Summer School at Giens in September 2002.

When reading the tutorial [15], it became clear that formalising Bertrand's postulate was possible. Many thanks go to Arkadii Slinko for setting up such an interesting web site.

Without the library of real analysis developed by Micaela Mayero and Olivier Desmettre, our work could not have been completed. A special thank goes to Olivier that adapted the library so to include some key properties that were needed in our formalisation.

Jean-Christophe Filliâtre gave us a very reactive support on WHY.

All the formalisation has been done using the neat user-interface PCOQ [13]. Yves Bertot was kind enough to spend some time to update PCOQ with the very latest version of COQ.

References

1. Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. Preprint, 2002, Available at `http://www.cse.iit.ac.in/primality.pdf`.
2. Martin Aigner and Günter M. Ziegler. *Proofs from THE BOOK*. Springer, 1998.
3. José C. Almeida and Laurent Théry. Correctness of the RSA algorithm. Coq contribution, 1999, Available at `http://coq.inria.fr/contribs/summary.html`.
4. Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A GMP program computing square roots and its proof within Coq. *Journal of Automated Reasoning*, 29(3–4), 2002.
5. Olga Caprotti and Martijn Oostdijk. Formal and Efficient Primality Proofs by Use of Computer Algebra Oracles. *Journal of Symbolic Computation*, 32(1):55–70, 2001.
6. Paul Erdös. Beweis eines Satzes von Tschebyschef. In *Acta Scientifica Mathematica*, volume 5, pages 194–198, 1932.
7. Jean-Christophe Filliâtre. Proof of Imperative Programs in Type Theory. In *TYPES '98*, volume 1657 of *LNCS*, 1998.
8. John Harrison. Floating Point Verification in HOL. In *Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *LNCS*, pages 186–199, 1995.
9. John Harrison and Laurent Théry. A Skeptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21(3):279–294, 1998.
10. C. Anthony R. Hoare. An Axiomatic Basis for Computer Programming. *Communication of the ACM*, 12(10):576–80, 583, October 1969.
11. Joe Hurd. Formal Verification of Probabilistic Algorithms. Phd Thesis, University of Cambridge, 2002.

12. Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, pages 147–149. Addison-Wesley, 1997.
13. PCoq. A Graphical User-interface to Coq, Available at `http://www-sop.inria.fr/lemme/pcoq/`.
14. Art Quaife. *Automated development of fundamental mathematical theories.* Automated reasoning series: 2. Kluwer, 1992.
15. Arkadii Slinko. Number Theory. Tutorial 5: Bertrand's Postulate. Available at `http://matholymp.com/tutorials/bertrand.pdf`.
16. Laurent Théry. A Tour of Formal Verification with Coq: Knuth's Algorithm for Prime Numbers. Research Report 4600, INRIA, 2002.

A The Complete Annotated Program

```
{ n > 0 }
begin
  a[0] := 2;
  m := 3;
  i := 1;
  while ((!i) < n) do
      {invariant
        (0 < i and i <= n)                                         and
        (a[i-1] < m and m < 2*a[i-1]) and Odd (m)                  and
        (forall k:int.
            (a[i-1] < k and k < m -> not(Prime(k))))               and
        (forall k:int. (0 <= k and k < i -> Prime(a[k])))          and
        (forall k:int. forall j:int.
            (0 <= k and k < j and j < i -> a[k] < a[j]))           and
        (forall k:int.
            (0 <= k and k <= a[i-1] and Prime(k)) -> In(k,a,0,i))
       variant (n-i, 2*a[i-1]-m) for lexZ }
    b := true;
    s := (sqr !m);
    j := 0;
    while (!b && a[!j] <= !s) do
       {invariant
         (if (b)
           then
             (forall k:int.
                (0 <= k and k < j -> not(Divides(a[k],m))))
           else Divides(a[j],m))                                   and
         (0 <= j and j < i)
        variant one(b)+i-j }
      if (mod !m a[!j]) = 0
      then b := false
      else j := !j + 1
    done;
    if (!b) then
      begin
        a[!i] := !m;
        i := !i + 1
      end;
    m := !m + 2
 done
end
{(forall k:int. (0 <= k and k < n -> Prime(a[k])))              and
 (forall k:int. forall j:int.
      (0 <= k and k < j and j < n -> a[k] < a[j]))              and
 (forall k:int.
      (0 <= k and k <= a[n-1] and Prime(k)) -> In(k,a,0,n))
}
```