A Tour of Formal Verification with Coq:Knuth's Algorithm for Prime Numbers

Laurent Théry

A Tour of Formal Verification with Coq: Knuth's Algorithm for Prime Numbers

Laurent Théry

N° 4600

October 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4600--FR+ENG

A Tour of Formal Verification with Coq: Knuth's Algorithm for Prime Numbers

Laurent Théry

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lemme

Rapport de recherche n° 4600 — October 2002 — 17 pages

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 mechanized in the COQ proof system.

Key-words: Prime numbers, Bertrand's Postulate, Program Verification, Coq.

Verification de Programme en Coq: L'Algorithme de Knuth pour les Nombres Premiers

Résumé : Nous nous intéressons à l'algorithme donné par Knuth dans son livre ”The Art of Computer Programming” pour calculer les n premiers nombres premiers et à sa correction. De façon assez surprenante, la vérification formelle d'un tel algorithme met en jeu un éventail assez large de techniques de vérification. Nous présentons comment cette vérification a été menée à bien dans le système Coq.

Mots-clés : Nombres premiers, Conjecture de Bertrand, Vérification de programme, Coq.

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 arithmetics 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. An illustration of the difficulty of verifying such algorithms can be found in [2].

In this paper we are interested in an algorithm given by Knuth in his book "The Art of Computer Programming" [9]. This algorithm takes an integer n as an argument and returns the list of the first n prime numbers. The correctness of this algorithm relies on a deep property of prime numbers called Bertrand's postulate. The property, first conjectured by Bertrand and proved by Chebischev, 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 to formally prove Bertrand's postulate. To do so we follow the proof given in [11]. The original idea of this elementary proof is due to Paul Erdös [4]. 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 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 is simple to define in a prover. Natural numbers are usually defined using Peano representation. For example in Coq, we have:

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

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

The fact that *a divides b* is written $divides(a, b)$. 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 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 one states that a number n is prime if all the prime numbers less than $\sqrt{n}$ do not divide it. The bound of $\sqrt{n}$ comes from the fact that if n is composite $n = pq$ then one element of the product either p or q is less than $\sqrt{n}$. This second property is stated as:

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

3 Knuth's Algorithm

To express the algorithm given by Knuth and to state its correctness, we are using the Why tool [5]. The tool takes an annotated programs with logical assertions à la Hoare [8] 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. We are now going to 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
```

From the declaration, `n` is a variable whose value cannot be modified. `a` is an array that should contain eventually the first `n` primes. end. `m`, `s`, `i` and `j` are modifiable integer variables. Finally `b` is a modifiable boolean variable. To write the algorithm, we need two extra functions on integer numbers:

```
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 is in

```
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

charge of filling the array `a` with prime numbers. For this, it uses a candidate prime number `m`. The boolean variable `b` is in charge of telling 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 is in charge of checking the primality of the value of `m`. For this it uses the property *primeDef$_1$*. The test `(mod !m a[!j])=0` checks 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:

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

the correctness of the program would be a direct consequence of the two properties *prime2Odd* and *primeDef₁*. 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 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 $\texttt{a[i-1]}^2$. 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 $2\texttt{a[i-1]} \leq \texttt{a[i-1]}^2$, 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* → 1 and *false* → 0. It is used to express some termination property 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 $\texttt{i} \leq \texttt{k} < \texttt{j}$ such that $\texttt{a[k]} = \texttt{n}$
The predicates `Prime`, `Odd` and `Divides` are the usual predicates on integers.

4.1 Pre and Post Conditions

The only precondition for the program is:

```
      { 0<n }
```

It is needed to legitimate the statement `a[0]=2`. The post condition 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`, `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 get 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 haven't yet found a divisor to `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 Figure 2

```
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

{ 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))
}
```

Fig. 2. The Complete Annotated Program

5 Bertrand's postulate

Running the Why tool on the annotated program given in Figure 2 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. If this section, we show how it is possible to get such a proof in a prover like Coq. This proof is a direct encoding of the proof presented in [11]. Still our presentation is slightly different and likely to be easier to formalize in a prover.

5.1 Divisibility and Primality Test Functions

Divisibility and primality are two key notions on which we need to be able to compute. For this we turn these predicates into test functions. The divisibility test $d(a,b)$ is defined as follows:

$d(a,b) = 1$ if b divides a
$d(a,b) = 0$ otherwise

The primality test is defined similarly:

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

5.2 Quotient

Another function that has to be defined is the quotient of two numbers. The quotient of a and b is written a/b and is uniquely defined with the following property:

Theorem *divUnique*: $\forall a, b, q\colon \mathbb{N}.\, 0 < b \wedge q * b \leq a < (q+1) * b \Rightarrow q = a/b$.

For example, $5/2$ gives 2. The quotient and the divisibility test are related by the following property:

Theorem *divDiracDiv*: $\forall a, b\colon \mathbb{N}.\, 0 < a \wedge 0 < b \Rightarrow a/b = \sum_{1 \leq i \leq a} d(i,b)$.

This property holds because the multiples of b in the interval $[1, \ldots, a]$ are b, $2b$, …, $(a/b)b$.

5.3 Maximal exponent

A complement to the divisibility test is the function that computes the maximal exponent r such that b^r divides a. The maximal exponent of b that divides a is written $max^\bullet(a,b)$ and is uniquely defined by the following theorem:

Theorem *powerDivInv*:
$\forall a, b, r\colon \mathbb{N}.\, 0 < a \wedge 1 < b \Rightarrow \mathit{divides}(b^r, a) \wedge \neg \mathit{divides}(b^{r+1}, a) \Rightarrow r = max^\bullet(a,b)$.

For prime numbers, the maximal exponent of a product is the sum of the maximal exponents:

Theorem $powerDivMultPrime$:
$\forall a, b, p \colon \mathbb{N}.\, 0 < a \wedge 0 < b \wedge prime(p) \Rightarrow max^\bullet(ab, p) = max^\bullet(a, p) + max^\bullet(b, p)$.

This is a consequence of the fact that if a prime number p divides a product ab it must divide a or b. Using the maximal exponent we can state the factorisation theorem:

Theorem $factorisation$: $\forall m \colon \mathbb{N}.\, 0 < n \Rightarrow n = \prod_{1 \leq i} 1_p(i)^{max^\bullet(n,i)}$.

The divisibility test and the maximal exponent are related by the following theorem:

Theorem $divDiracPowerDiv$: $\forall a, b \colon \mathbb{N}.\, 0 < a \wedge 1 < b \Rightarrow max^\bullet(a, b) = \sum_{1 \leq i} d(a, b^i)$.

It is proved using the fact that if b^i divides a, then for all j smaller than i, b^j also divides a. It is also possible to relate quotient and maximal exponent for the factorial:

Theorem $powerDivFactPrime$: $\forall p, n \colon \mathbb{N}.\, 0 < n \wedge prime(p) \Rightarrow max^\bullet(n!, p) = \sum_{1 \leq i} n/p^i$.

To prove this property, we first apply iteratively the theorem *powerDivMultPrime* to get:

$$max^\bullet(n!, p) = \sum_{1 \leq j \leq n} max^\bullet(j, p)$$

Applying the theorem *divDiracPowerDiv* we get:

$$max^\bullet(n!, p) = \sum_{1 \leq j \leq n} \sum_{1 \leq i} d(j, p^i)$$

Commuting the two sums we get:

$$max^\bullet(n!, p) = \sum_{1 \leq i} \sum_{1 \leq j \leq n} d(j, p^i)$$

The theorem *divDiracDiv* ends the proof:

$$max^\bullet(n!, p) = \sum_{1 \leq i} n/p^i$$

5.4 Binomial Coefficients

A central 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 and a lower bound for $\binom{2n}{n}$. To define binomial coefficients, we use Pascal's triangle and derive the following equivalent definition:

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

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

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

They are consequences of the theorem *binomialFact*.

In order to establish a bound on $\binom{2n}{n}$, we need to prove the following property for the maximal exponent of each prime number:

Theorem *powerDivBinomial*:
$\forall n, p \colon \mathbb{N}.\, 0 < n \wedge \mathit{prime}(p) \Rightarrow \mathit{max}^{\bullet}(\binom{2n}{n}, p) = \sum_{1 \leq i}((2n)/p^i - 2(n/p^i))$.

To prove it, we start from the equality;

$$2n! = \binom{2n}{p} n! n!$$

Applying the theorem *powerDivMultPrime* we get:

$$\mathit{max}^{\bullet}(2n!, p) = \mathit{max}^{\bullet}(\binom{2n}{n}, p) + 2\mathit{max}^{\bullet}(n!, p)$$

The theorem *powerDivFactPrime* gives us the expected equality.
From this equality, it is possible to derive three corollaries. The first corollary gives an upper bound for $p^{\mathit{max}^{\bullet}(\binom{2n}{n}, p)}$:

Theorem *powerDivBinomial*$_1$: $\forall n, p \colon \mathbb{N}.\, 0 < n \wedge \mathit{prime}(p) \Rightarrow p^{\mathit{max}^{\bullet}(\binom{2n}{n}, p)} \leq 2n$.

To prove it, we first notice that

$$0 \leq (2n)/p^i - 2(n/p^i) \leq 1$$

In particular we are sure that the value is 0 if we have p^i strictly greater than $2n$. This means that if j is the largest integer such that $p^j \leq 2n$, we have:

$$\mathit{max}^{\bullet}(\binom{2n}{n}, p) = \sum_{1 \leq i \leq j}((2n)/p^i - 2(n/p^i)) \leq \sum_{1 \leq i \leq j} 1 = j$$

Using the monotony of the power function we get

$$p^{\mathit{max}^{\bullet}(\binom{2n}{n}, p)} \leq p^j \leq 2n$$

The second corollary gives an upper bound for $\mathit{max}^{\bullet}(\binom{2n}{n}, p)$ in a special case:

Theorem *powerDivBinomial*$_2$: $\forall n, p \colon \mathbb{N}.\, 0 < n \wedge 2n < p^2 \wedge \mathit{prime}(p) \Rightarrow \mathit{max}^{\bullet}(\binom{2n}{n}, p) \leq 1$.

The condition $2n < p^2$ indicates that the largest j such that $p^j \leq 2n$ is 1 or 0 so:

$$\mathit{max}^{\bullet}(\binom{2n}{n}, p) = (2n)/p - 2(n/p) \leq 1$$

The last corollary gives the exact value of $\mathit{max}^{\bullet}(\binom{2n}{n}, p)$ in a special case:

Theorem *powerDivBinomial*$_3$:
$\forall n, p \colon \mathbb{N}.\, 2 < n \wedge 2n < 3p \wedge p \leq n \wedge \mathit{prime}(p) \Rightarrow \mathit{max}^{\bullet}(\binom{2n}{n}, p) = 0$.

The conditions $2 < n$ and $2n < 3p$ indicates that n is at least 3 then p is at least 3, this means that $2n < 3p \leq p^2$. We can apply the previous corollary and have:

$max^\bullet(\binom{2n}{n}, p) = (2n)/p - 2(n/p)$

Using the conditions $2n < 3p$ and $p \leq n$, we get $2p \leq n < 3p$.

Applying the theorem *divUnique* we get that $2n/p = 2$.

Using the conditions $2n < 3p$ and $p \leq n$, we get $1p \leq n < 2p$.

Applying the theorem *divUnique* we get that $n/p = 1$.

It follows that $max^\bullet(\binom{2n}{n}, p) = 2 - 2.1 = 0$.

5.5 Binomial Theorem

Another ingredient to get the final bound on $\binom{2n}{n}$ is the binomial theorem:

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

We use this theorem in the special case where $a = b = 1$:

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

This last theorem gives us an upper bound for $\binom{2n+1}{n}$:

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

We have:

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

Keeping only the two middle terms of the sum, we get:

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

Using the theorem *binomialComp*, we get :

$$2.\binom{2n+1}{n} \leq 2^{2n+1}$$

Simplifying by 2 on both sides we get the expected result. In the same way we get a lower bound for $\binom{2n}{n}$:

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

Using the theorem *binomial*$_2$, we have:

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

Taking apart the first and the last term gives:

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

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

$\binom{2n}{i} \leq \binom{2n}{n}$ for $1 \leq i \leq 2n-1$

Using these two bounds we have:

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

5.6 Product of primes

The last theorem gives us a lower bound for $\binom{2n}{n}$. To be able to establish an upper bound, we need to get an upper bound on the product of primes:

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

It is proved by 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 property holds for n. If n is even, we have:

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

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 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 prime of the product divides $\binom{2m+1}{m}$. So we have:

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

The theorem *binomialEven* gives an upper bound for the binomial:

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

5.7 Upper Bound

The theorem *binomialEven* gives us a lower bound for $\binom{2n}{n}$. The upper bound is proved under two assumptions: there is no prime between n and $2n$ and $128 \leq n$.

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

To prove this result, we first use the theorem *factorisation*:

$$\binom{2n}{n} = \prod_{1 \leq i} 1_p(i)^{max^{\bullet}(\binom{2n}{n}, i)}$$

We cannot have prime numbers in the product greater than $2n$. So we can restrict the product to $2n$

$$\binom{2n}{n} = \prod_{i \leq 2n} 1_p(i)^{max^{\bullet}(\binom{2n}{n},i)}$$

Since there is no prime between n and $2n$ we also have:

$$\binom{2n}{n} = \prod_{i \leq n} 1_p(i)^{max^{\bullet}(\binom{2n}{n},i)}$$

From the corollary *powerDivBinomial*$_3$ we can further restrict the product:

$$\binom{2n}{n} = \prod_{i \leq 2n/3} 1_p(i)^{max^{\bullet}(\binom{2n}{n},i)}$$

Now we can split the sum in two:

$$\binom{2n}{n} = (\prod_{i \leq \sqrt{2n}} 1_p(i)^{max^{\bullet}(\binom{2n}{n},i)}) \quad (\prod_{\sqrt{2n} < i \leq 2n/3} 1_p(i)^{max^{\bullet}(\binom{2n}{n},i)})$$

The corollary *powerDivBinomial*$_2$ shows that all the maximal exponents of the second product are equal to 1:

$$\binom{2n}{n} = (\prod_{i \leq \sqrt{2n}} 1_p(i)^{max^{\bullet}(\binom{2n}{n},i)}) \quad (\prod_{\sqrt{2n} < i \leq 2n/3} 1_p(i))$$

We can loosen the second product and apply the theorem *prodPrimeLt*

$$\binom{2n}{n} \leq (\prod_{i \leq \sqrt{2n}} 1_p(i)^{max^{\bullet}(\binom{2n}{n},i)}) \quad (\prod_{i \leq 2n/3} 1_p(i)) < (\prod_{i \leq \sqrt{2n}} 1_p(i)^{max^{\bullet}(\binom{2n}{n},i)}) \quad 4^{2n/3}$$

The corollary *powerDivBinomial*$_2$ gives an upper bound $2n$ for each term in the product. We are left with getting an estimate of the numbers of prime less than $\sqrt{2n}$. We can use the fact that for $16 \leq p$, there is no more than $p/2 - 1$ primes less than p (even number except 2 are not prime). For $p = \sqrt{2n} \geq \sqrt{256} = 16$, there is no more than $\sqrt{2n}/2 - 1$ prime less than $\sqrt{2n}$. This gives us an upper bound for the product:

$$\binom{2n}{n} < 2n^{\sqrt{2n}/2-1} 4^{2n/3}$$

5.8 Function Analysis

We are now very close to prove Bertrand's postulate. The theorems *binomialEven* and *upperBound* give a lower and an upper bound for $(2n)\binom{2n}{n}$ respectively:

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

In this inequation the operation $\sqrt{\ }$ and $/$ are functions over natural numbers, i.e $\sqrt{6} = 2$ and $7/2 = 3$. Because of monotonicity, the equation should also hold for functions over the reals. This means that if we are able to prove that for x real and $128 \leq x$:

$$2x^{\frac{\sqrt{2x}}{2}} 4^{\frac{2x}{3}} < 4^x$$

the only way the theorem *upperBound* could hold 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, we first simplify it using properties of the power function:

$$2x^{\frac{\sqrt{2x}}{2}} < 4^{\frac{x}{3}}$$

Taking logarithm on both sides, we have:

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

Simplifying it again we get

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

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

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

If we compute the derivative of f we get:

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

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

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

6 Some Remarks on the Formal Development

The really difficult part of this formalisation was obviously the proof of Bertrand's postulate. We follow closely the steps described in [11]. Doing so we discover that the applicability of two properties should be restricted. For the theorem:

Theorem *powerDivBinomial*$_3$:
$\forall n, p\colon \mathbb{N}.\, 2 < n \wedge 2n < 3p \wedge p \leq n \wedge prime(p) \Rightarrow max^{\bullet}(\binom{2n}{n}, p) = 0.$

the condition $2 < n$ was missing. For the theorem:

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

the condition $128 \leq n$ was missing. It was wrongly stated in the proof 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.

Formalizing the analysis of the function f only requires basic notions about the exponential and logarithmic functions. It needs also the fact that a function that has a positive derivative is increasing. This is a consequence of the intermediate value theorem.

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

This theorems 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 $\neg prime(p)$. It means for example that a proof of $prime(11)$ is simply *primebCorrect(11)* since $primeb(11)$ evaluates to *true*. Using functions that can be evaluated inside Coq is interesting because not only it allows to automate proofs but also it generates very small proof objects. Following the same idea we have a function *checkPostulate* with the associated theorem of correctness:

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

Then to prove the following theorem:

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

we use directly the previous theorem with a proof of $checkPostulate(128) = true$. Since $checkPostulate(128)$ evaluates to *true*, a proof of $checkPostulate(128) = true$ is the same than a proof of $true = true$, a consequence of the reflexivity of equality. A complete proof of the theorem *postulateCorrect128* is $postulateCorrect(128, reflEqual(bool, true))$.

The full development is available at
`ftp://ftp-sop.inria.fr/lemme/Laurent.Thery/Bertrand/index.html`.
It took us two weeks of intensive work to complete. It is 6000 line long. The actual proof of the 18 verification conditions is 1000 line long of which 800 are just the statements of the conditions.

7 Acknowledgments

This formalisation was motivated by the preparation of an introductive lecture on Formalised Mathematics that was given at the Types Summer School at Giens in September 2002.

When reading the nice tutorial [11], it became clear that formalising Bertrand's postulate was possible. Many thanks goes to Arkadii Slinko for setting up such an interesting web site.

Without the library of real analysis developed by Micaela Mayero and Olivier Desmettre, this 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 this formalisation.

Jean-Christophe Filliâtre gives us a very reactive support on Why.

All the formalisation has been done using the very neat user-interface Pcoq [10]. Yves Bertot was kind enough for spending some time to update Pcoq with the very latest version of Coq.

8 Conclusions

We believe that this proof is a nice and relatively simple example of both the complexity and diversity of program verification. We took a very simple program and trying to prove it correct from basic principles took us on quite a long tour. One of the verification condition requires a deep property of number theory, namely Bertrand's postulate. To prove such a theorem we had to do some work in discrete mathematics. This was no surprise. More surprisingly the final steps require to transfer the problem into real analysis. Lately 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. Proving this property has been done 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 [1] 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 in [7] would be to delegate to an external program the search for all the necessary primes and use Coq only to check the result. Note that a similar method has already been used to get the proof of relatively large prime numbers in Coq [3].

Before this proof, the only formalised proof we were aware of that was exhibiting a similar diversity of verification techniques is [6]. The fact that it is possible to carry such verifications in a single system shows how generic provers in higher-order logics are. Having an expressive logics makes them suitable for all different kind of verifications.

References

1. Martin Aigner and Günter M. Ziegler. *Proofs from THE BOOK*. Springer, 1998.
2. Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A GMP program computing square roots and its proof within Coq. *Journal of Automated Reasoning*, 2003. To appear.
3. 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.
4. Paul Erdös. Beweis eines Satzes von Tschebyschef. In *Acta Scientifica Mathematica*, volume 5, pages 194–198, 1932.
5. J.-C. Filliâtre. Proof of Imperative Programs in Type Theory. In *TYPES '98*, volume 1657 of *LNCS*, 1998.
6. John Harrison. Floating Point Verification in HOL. In *Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *LNCS*, pages 186–199, 1995.
7. John Harrison and Laurent Théry. A Skeptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21(3):279–294, 1998.
8. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communication of the ACM*, 12(10):576–80, 583, October 1969.
9. Donald E. Knuth. *The Art of Computer Programming, : Fundamental Algorithms*, pages 147–149. Addison-Wesley, 1997.
10. Pcoq. A Graphical User-interface to Coq, Available at `http://www-sop.inria.fr/lemme/pcoq/`.
11. Arkadii Slinko. Number Theory. Tutorial 5: Bertrand's Postulate. Available at `http://matholymp.com/tutorials/bertrand.pdf`.

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399