

A Formalization of Pratt's Primality Certificates

By Simon Wimmer and Lars Noschinski

May 26, 2024

Abstract

In 1975, Pratt introduced a proof system for certifying primes [1]. He showed that a number p is prime iff a primality certificate for p exists. By showing a logarithmic upper bound on the length of the certificates in size of the prime number, he concluded that the decision problem for prime numbers is in NP. This work formalizes soundness and completeness of Pratt's proof system as well as an upper bound for the size of the certificate.

Contents

1	Pratt's Primality Certificates	1
1.1	Soundness	2
1.2	Completeness	4
1.3	Efficient modular exponentiation	12
1.4	Executable certificate checker	14
1.5	Proof method setup	19
1.6	Code generation for Pratt certificates	20

1 Pratt's Primality Certificates

theory *Pratt-Certificate*

imports

Complex-Main

Lehmer.Lehmer

begin

This work formalizes Pratt's proof system as described in his article "Every Prime has a Succinct Certificate"[1].

The proof system makes use of two types of predicates:

- $\text{Prime}(p)$: p is a prime number
- (p, a, x) : $\forall q \in \text{prime-factors}(x). [a^{\wedge}((p - 1) \text{ div } q) \neq 1] \pmod{p}$

We represent these predicates with the following datatype:

datatype *pratt* = *Prime nat* | *Triple nat nat nat*

Pratt describes an inference system consisting of the axiom $(p, a, 1)$ and the following inference rules:

- R1: If we know that (p, a, x) and $[a \wedge ((p - 1) \text{ div } q) \neq 1] \pmod{p}$ hold for some prime number q we can conclude (p, a, qx) from that.
- R2: If we know that $(p, a, p - 1)$ and $[a \wedge (p - 1) = 1] \pmod{p}$ hold, we can infer $\text{Prime}(p)$.

Both rules follow from Lehmer's theorem as we will show later on.

A list of predicates (i.e., values of type *pratt*) is a *certificate*, if it is built according to the inference system described above. I.e., a list $x \# xs$ is a certificate if xs is a certificate and x is either an axiom or all preconditions of x occur in xs .

We call a certificate xs a *certificate for* p , if *Prime* p occurs in xs .

The function *valid-cert* checks whether a list is a certificate.

```
fun valid-cert :: pratt list  $\Rightarrow$  bool where
  valid-cert [] = True
| R2: valid-cert (Prime p # xs)  $\longleftrightarrow$   $1 < p \wedge$  valid-cert xs
   $\wedge (\exists a . [a \wedge (p - 1) = 1] \pmod{p} \wedge \text{Triple } p \ a \ (p - 1) \in \text{set } xs)$ 
| R1: valid-cert (Triple p a x # xs)  $\longleftrightarrow$   $p > 1 \wedge 0 < x \wedge$  valid-cert xs  $\wedge (x=1 \vee$ 
   $(\exists q \ y. x = q * y \wedge \text{Prime } q \in \text{set } xs \wedge \text{Triple } p \ a \ y \in \text{set } xs$ 
   $\wedge [a \wedge ((p - 1) \text{ div } q) \neq 1] \pmod{p}))$ 
```

We define a function *size-cert* to measure the size of a certificate, assuming a binary encoding of numbers. We will use this to show that there is a certificate for a prime number p such that the size of the certificate is polynomially bounded in the size of the binary representation of p .

```
fun size-pratt :: pratt  $\Rightarrow$  real where
  size-pratt (Prime p) =  $\log 2 \ p$  |
  size-pratt (Triple p a x) =  $\log 2 \ p + \log 2 \ a + \log 2 \ x$ 
```

```
fun size-cert :: pratt list  $\Rightarrow$  real where
  size-cert [] =  $0$  |
  size-cert ( $x \# xs$ ) =  $1 + \text{size-pratt } x + \text{size-cert } xs$ 
```

1.1 Soundness

In Section 1 we introduced the predicates $\text{Prime}(p)$ and (p, a, x) . In this section we show that for a certificate every predicate occurring in this certificate holds. In particular, if $\text{Prime}(p)$ occurs in a certificate, p is prime.

lemma *prime-factors-one* [*simp*]: **shows** *prime-factors* (*Suc 0*) = $\{\}$

```

using prime-factorization-1 [where ?'a = nat] by simp

lemma prime-factors-of-prime: fixes p :: nat assumes prime p shows prime-factors
p = {p}
  using assms by (fact prime-prime-factors)

definition pratt-triple :: nat ⇒ nat ⇒ nat ⇒ bool where
  pratt-triple p a x ⇔ x > 0 ∧ (∀ q ∈ prime-factors x. [a ^ ((p - 1) div q) ≠ 1]
(mod p))

lemma pratt-triple-1: p > 1 ⇒ x = 1 ⇒ pratt-triple p a x
  by (auto simp: pratt-triple-def)

lemma pratt-triple-extend:
  assumes prime q pratt-triple p a y
    p > 1 x > 0 x = q * y [a ^ ((p - 1) div q) ≠ 1] (mod p)
  shows pratt-triple p a x
proof -
  have prime-factors x = insert q (prime-factors y)
    using assms by (simp add: prime-factors-product prime-prime-factors)
  also have ∀ r ∈ ... [a ^ ((p - 1) div r) ≠ 1] (mod p)
    using assms by (auto simp: pratt-triple-def)
  finally show ?thesis using assms
    unfolding pratt-triple-def by blast
qed

lemma pratt-triple-imp-prime:
  assumes pratt-triple p a x p > 1 x = p - 1 [a ^ (p - 1) = 1] (mod p)
  shows prime p
  using lehmers-theorem[of p a] assms by (auto simp: pratt-triple-def)

theorem pratt-sound:
  assumes 1: valid-cert c
  assumes 2: t ∈ set c
  shows (t = Prime p → prime p) ∧
    (t = Triple p a x → ((∀ q ∈ prime-factors x . [a ^ ((p - 1) div q) ≠ 1]
(mod p)) ∧ 0 < x))
  using assms
proof (induction c arbitrary: p a x t)
  case Nil then show ?case by force
  next
  case (Cons y ys)
  { assume y = Triple p a x x = 1
    then have (∀ q ∈ prime-factors x . [a ^ ((p - 1) div q) ≠ 1] (mod p)) ∧ 0 < x
  by simp
  }
  moreover
  { assume x = y: y = Triple p a x x = 1
    hence x > 0 using Cons.prem by auto
  }

```

```

obtain  $q z$  where  $x=q*z$   $Prime\ q \in\ set\ ys \wedge\ Triple\ p\ a\ z \in\ set\ ys$ 
and  $cong:[a^{\wedge}(p-1)\ div\ q] \neq 1 \pmod p$  using  $Cons.prem\ x-y$  by
auto
then have  $factors-IH:(\forall\ r \in\ prime-factors\ z . [a^{\wedge}(p-1)\ div\ r] \neq 1 \pmod p)$ 
prime\ q\ z > 0
using  $Cons.IH\ Cons.prem\ \langle x > 0 \rangle\ \langle y = Triple\ p\ a\ x \rangle$ 
by force+
then have  $prime-factors\ x = prime-factors\ z \cup \{q\}$  using  $\langle x = q*z \rangle\ \langle x > 0 \rangle$ 
by (simp\ add:\ prime-factors-product\ prime-factors-of-prime)
then have  $(\forall\ q \in\ prime-factors\ x . [a^{\wedge}(p-1)\ div\ q] \neq 1 \pmod p) \wedge 0 < x$ 
using  $factors-IH\ cong$  by (simp\ add:\ \langle x > 0 \rangle)
}
ultimately have  $y-Triple:y = Triple\ p\ a\ x \implies (\forall\ q \in\ prime-factors\ x .$ 
 $[a^{\wedge}(p-1)\ div\ q] \neq 1 \pmod p) \wedge 0 < x$  by
linarith
{ assume  $y = Prime\ p\ p > 2$  then
obtain  $a$  where  $a:[a^{\wedge}(p-1) = 1] \pmod p$   $Triple\ p\ a\ (p-1) \in\ set\ ys$ 
using  $Cons.prem\ by\ auto$ 
then have  $Bier:(\forall\ q \in\ prime-factors\ (p-1) . [a^{\wedge}(p-1)\ div\ q] \neq 1 \pmod p)$ 
using  $Cons.IH\ Cons.prem(1)$  by (simp\ add:y(1))
then have  $prime\ p$  using  $lehmers-theorem[OF\ -a(1)]\ \langle p > 2 \rangle$  by fastforce
}
moreover
{ assume  $y = Prime\ p\ p = 2$  hence  $prime\ p$  by simp }
moreover
{ assume  $y = Prime\ p$  then have  $p > 1$  using  $Cons.prem\ by\ simp$  }
ultimately have  $y-Prime:y = Prime\ p \implies prime\ p$  by linarith

show ?case
proof (cases\ t \in\ set\ ys)
case True
show ?thesis using  $Cons.IH[OF\ -True]\ Cons.prem(1)$  by (cases\ y) auto
next
case False
thus ?thesis using  $Cons.prem(2)\ y-Prime\ y-Triple$  by force
qed
qed

```

corollary *pratt-primeI*:

```

assumes  $valid-cert\ xs\ Prime\ p \in\ set\ xs$ 
shows  $prime\ p$ 
using  $pratt-sound[OF\ assms]$  by simp

```

1.2 Completeness

In this section we show completeness of Pratt's proof system, i.e., we show that for every prime number p there exists a certificate for p . We also give an upper bound for the size of a minimal certificate

The prove we give is constructive. We assume that we have certificates for

all prime factors of $p - 1$ and use these to build a certificate for p from that. It is important to note that certificates can be concatenated.

lemma *valid-cert-appendI*:
assumes *valid-cert r*
assumes *valid-cert s*
shows *valid-cert (r @ s)*
using *assms*
proof (*induction r*)
case (*Cons y ys*) **then show** *?case* **by** (*cases y*) *auto*
qed *simp*

lemma *valid-cert-concatI*: $(\forall x \in \text{set } xs . \text{valid-cert } x) \implies \text{valid-cert } (\text{concat } xs)$
by (*induction xs*) (*auto simp add: valid-cert-appendI*)

lemma *size-pratt-le*:
fixes *d::real*
assumes $\forall x \in \text{set } c . \text{size-pratt } x \leq d$
shows *size-cert c ≤ length c * (1 + d)* **using** *assms*
by (*induction c*) (*simp-all add: algebra-simps*)

fun *build-fpc* :: *nat ⇒ nat ⇒ nat ⇒ nat list ⇒ pratt list* **where**
build-fpc p a r [] = [Triple p a r] |
build-fpc p a r (y # ys) = Triple p a r # build-fpc p a (r div y) ys

The function *build-fpc* helps us to construct a certificate for p from the certificates for the prime factors of $p - 1$. Called as *build-fpc p a (p - 1) qs* where $qs = q_1 \dots q_n$ is prime decomposition of $p - 1$ such that $q_1 \dots q_n = p - 1$, it returns the following list of predicates:

$$(p, a, p - 1), (p, a, \frac{p - 1}{q_1}), (p, a, \frac{p - 1}{q_1 q_2}), \dots, (p, a, \frac{p - 1}{q_1 \dots q_n}) = (p, a, 1)$$

I.e., if there is an appropriate a and a certificate rs for all prime factors of p , then we can construct a certificate for p as

$$\text{Prime } p \# \text{build-fpc } p \ a \ (p - 1) \ qs \ @ \ rs$$

The following lemma shows that *build-fpc* extends a certificate that satisfies the preconditions described before to a correct certificate.

lemma *correct-fpc*:
assumes *valid-cert xs p > 1*
assumes *prod-list qs = r r ≠ 0*
assumes $\forall q \in \text{set } qs . \text{Prime } q \in \text{set } xs$
assumes $\forall q \in \text{set } qs . [a \wedge (p - 1) \text{ div } q \neq 1] \pmod{p}$
shows *valid-cert (build-fpc p a r qs @ xs)*
using *assms*
proof (*induction qs arbitrary: r*)
case *Nil* **thus** *?case* **by** *auto*

```

next
  case (Cons y ys)
  have prod-list ys = r div y using Cons.premis by auto
  then have T-in: Triple p a (prod-list ys) ∈ set (build-fpc p a (r div y) ys @ xs)
    by (cases ys) auto

  have valid-cert (build-fpc p a (r div y) ys @ xs)
    using Cons.premis by (intro Cons.IH) auto
  then have valid-cert (Triple p a r # build-fpc p a (r div y) ys @ xs)
    using ⟨r ≠ 0⟩ T-in Cons.premis by auto
  then show ?case by simp
qed

```

```

lemma length-fpc:
  length (build-fpc p a r qs) = length qs + 1 by (induction qs arbitrary: r) auto

```

```

lemma div-gt-0:
  fixes m n :: nat assumes m ≤ n 0 < m shows 0 < n div m
proof -
  have 0 < m div m using ⟨0 < m⟩ div-self by auto
  also have m div m ≤ n div m using ⟨m ≤ n⟩ by (rule div-le-mono)
  finally show ?thesis .
qed

```

```

lemma size-pratt-fpc:
  assumes a ≤ p r ≤ p 0 < a 0 < r 0 < p prod-list qs = r
  shows ∀ x ∈ set (build-fpc p a r qs) . size-pratt x ≤ 3 * log 2 p using assms
proof (induction qs arbitrary: r)
  case Nil
  then have log 2 a ≤ log 2 p log 2 r ≤ log 2 p by auto
  then show ?case by simp
next
  case (Cons q qs)
  then have log 2 a ≤ log 2 p log 2 r ≤ log 2 p by auto
  then have log 2 a + log 2 r ≤ 2 * log 2 p by arith
  moreover have r div q > 0 using Cons.premis by (fastforce intro: div-gt-0)
  moreover hence prod-list qs = r div q using Cons.premis(6) by auto
  moreover have r div q ≤ p using ⟨r ≤ p⟩ div-le-dividend[of r q] by linarith
  ultimately show ?case using Cons by simp
qed

```

```

lemma concat-set:
  assumes ∀ q ∈ qs . ∃ c ∈ set cs . Prime q ∈ set c
  shows ∀ q ∈ qs . Prime q ∈ set (concat cs)
  using assms by (induction cs) auto

```

```

lemma p-in-prime-factorsE:
  fixes n :: nat
  assumes p ∈ prime-factors n 0 < n

```

```

obtains  $2 \leq p \wedge p \leq n \wedge p \text{ dvd } n \wedge \text{prime } p$ 
proof
  from assms show prime p by auto
  then show  $2 \leq p$  by (auto dest: prime-gt-1-nat)

  from assms show p dvd n by auto
  then show  $p \leq n$  using  $\langle 0 < n \rangle$  by (rule dvd-imp-le)
qed

lemma prime-factors-list-prime:
  fixes  $n :: \text{nat}$ 
  assumes prime n
  shows  $\exists qs. \text{prime-factors } n = \text{set } qs \wedge \text{prod-list } qs = n \wedge \text{length } qs = 1$ 
  using assms by (auto simp add: prime-factorization-prime intro: exI [of - [n]])

lemma prime-factors-list:
  fixes  $n :: \text{nat}$  assumes  $3 < n \wedge \neg \text{prime } n$ 
  shows  $\exists qs. \text{prime-factors } n = \text{set } qs \wedge \text{prod-list } qs = n \wedge \text{length } qs \geq 2$ 
  using assms
proof (induction n rule: less-induct)
  case (less n)
    obtain  $p$  where  $p \in \text{prime-factors } n$  using  $\langle n > 3 \rangle$  prime-factors-elim by
    force
    then have  $2 \leq p \wedge p \leq n \wedge p \text{ dvd } n \wedge \text{prime } p$ 
    using  $\langle 3 < n \rangle$  by (auto elim: p-in-prime-factorsE)
    { assume  $n \text{ div } p > 3 \wedge \neg \text{prime } (n \text{ div } p)$ 
      then obtain  $qs$ 
        where  $\text{prime-factors } (n \text{ div } p) = \text{set } qs \wedge \text{prod-list } qs = (n \text{ div } p) \wedge \text{length } qs \geq$ 
        2
        using  $p'$  by atomize-elim (auto intro: less simp: div-gt-0)
      moreover
        have  $\text{prime-factors } (p * (n \text{ div } p)) = \text{insert } p (\text{prime-factors } (n \text{ div } p))$ 
        using  $\langle 3 < n \rangle \langle 2 \leq p \rangle \langle p \leq n \rangle \langle \text{prime } p \rangle$ 
        by (auto simp: prime-factors-product div-gt-0 prime-factors-of-prime)
        ultimately
        have  $\text{prime-factors } n = \text{set } (p \# qs) \wedge \text{prod-list } (p \# qs) = n \wedge \text{length } (p \# qs) \geq$ 
        2
        using  $\langle p \text{ dvd } n \rangle$  by simp-all
        hence ?case by blast
      }
    moreover
    { assume  $\text{prime } (n \text{ div } p)$ 
      then obtain  $qs$ 
        where  $\text{prime-factors } (n \text{ div } p) = \text{set } qs \wedge \text{prod-list } qs = (n \text{ div } p) \wedge \text{length } qs =$ 
        1
        using prime-factors-list-prime by blast
      moreover
        have  $\text{prime-factors } (p * (n \text{ div } p)) = \text{insert } p (\text{prime-factors } (n \text{ div } p))$ 
        using  $\langle 3 < n \rangle \langle 2 \leq p \rangle \langle p \leq n \rangle \langle \text{prime } p \rangle$ 
    }
  }

```

```

    by (auto simp: prime-factors-product div-gt-0 prime-factors-of-prime)
    ultimately
    have prime-factors n = set (p # qs) prod-list (p # qs) = n length (p#qs) ≥
2
      using ⟨p dvd n⟩ by simp-all
      hence ?case by blast
    } note case-prime = this
  moreover
  { assume n div p = 1
    hence n = p using ⟨n>3⟩ using One-leq-div[OF ⟨p dvd n⟩] p'(2) by force
    hence ?case using ⟨prime p⟩ ⟨¬ prime n⟩ by auto
  }
  moreover
  { assume n div p = 2
    hence ?case using case-prime by force
  }
  moreover
  { assume n div p = 3
    hence ?case using p' case-prime by force
  }
  ultimately show ?case using p' div-gt-0[of p n] case-prime by fastforce

```

qed

lemma prod-list-ge:

```

  fixes xs::nat list
  assumes ∀ x ∈ set xs . x ≥ 1
  shows prod-list xs ≥ 1 using assms by (induction xs) auto

```

lemma sum-list-log:

```

  fixes b::real
  fixes xs::nat list
  assumes b: b > 0 b ≠ 1
  assumes xs:∀ x ∈ set xs . x ≥ b
  shows (∑ x←xs. log b x) = log b (prod-list xs)
  using assms
proof (induction xs)
  case Nil
  thus ?case by simp
  next
  case (Cons y ys)
  have real (prod-list ys) > 0 using prod-list-ge Cons.prem1 by fastforce
  thus ?case using log-mult Cons.prem1 Cons by simp

```

qed

lemma concat-length-le:

```

  fixes g :: nat ⇒ real
  assumes ∀ x ∈ set xs . real (length (f x)) ≤ g x
  shows length (concat (map f xs)) ≤ (∑ x←xs. g x) using assms

```


by (induction xs) force+

lemma *prime-gt-3-impl-p-minus-one-not-prime*:

fixes $p::nat$

assumes *prime* $p > 3$

shows \neg *prime* $(p - 1)$

proof

assume *prime* $(p - 1)$

have \neg *even* p **using** *assms* **by** (*simp add: prime-odd-nat*)

hence $2 \text{ dvd } (p - 1)$ **by** *presburger*

then obtain q **where** $p - 1 = 2 * q$..

then have $2 \in \text{prime-factors } (p - 1)$ **using** $\langle p > 3 \rangle$

by (*auto simp: prime-factorization-times-prime*)

thus *False* **using** *prime-factors-of-prime* $\langle p > 3 \rangle$ $\langle \text{prime } (p - 1) \rangle$ **by** *auto*

qed

We now prove that Pratt's proof system is complete and derive upper bounds for the length and the size of the entries of a minimal certificate.

theorem *pratt-complete'*:

assumes *prime* p

shows $\exists c. \text{Prime } p \in \text{set } c \wedge \text{valid-cert } c \wedge \text{length } c \leq 6 * \log 2 p - 4 \wedge (\forall x \in \text{set } c. \text{size-pratt } x \leq 3 * \log 2 p)$ **using** *assms*

proof (*induction p rule: less-induct*)

case (*less p*)

from $\langle \text{prime } p \rangle$ have $p > 1$ **by** (*rule prime-gt-1-nat*)

then consider $p = 2 \mid p = 3 \mid p > 3$ **by** *force*

thus *?case*

proof *cases*

assume [*simp*]: $p = 2$

have *Prime* $p \in \text{set } [\text{Prime } 2, \text{Triple } 2 \ 1 \ 1]$ **by** *simp*

thus *?case* **by** *fastforce*

next

assume [*simp*]: $p = 3$

let *?cert* = [*Prime* 3, *Triple* 3 2 2, *Triple* 3 2 1, *Prime* 2, *Triple* 2 1 1]

have *length* *?cert* $\leq 6 * \log 2 p - 4 \iff 3 \leq 2 * \log 2 3$ **by** *simp*

also have $2 * \log 2 3 = \log 2 (3^2 :: \text{real})$ **by** (*subst log-nat-power*) *simp-all*

also have $\dots = \log 2 9$ **by** *simp*

also have $3 \leq \log 2 9 \iff \text{True}$ **by** (*subst le-log-iff*) *simp-all*

finally show *?case*

by (*intro exI[where x = ?cert]*) (*simp add: cong-def*)

next

assume $p > 3$

have *qlp*: $\forall q \in \text{prime-factors } (p - 1). q < p$ **using** $\langle \text{prime } p \rangle$

by (*metis One-nat-def Suc-pred le-imp-less-Suc lessI less-trans p-in-prime-factorsE prime-gt-1-nat zero-less-diff*)

hence *factor-certs*: $\forall q \in \text{prime-factors } (p - 1). (\exists c. ((\text{Prime } q \in \text{set } c) \wedge (\text{valid-cert } c)$

$\wedge \text{length } c \leq 6 * \log 2 q - 4) \wedge (\forall x \in$

*set c. size-pratt $x \leq 3 * \log 2 q$)*
by (*auto intro: less.IH*)
obtain a where $a: [a \wedge (p - 1) = 1] \pmod{p} \wedge (\forall q. q \in \text{prime-factors } (p - 1) \rightarrow [a \wedge ((p - 1) \text{ div } q) \neq 1] \pmod{p})$ **and** *a-size: $a > 0 \ a < p$*
using *converse-lehmer[OF <prime p>]* **by** *blast*

have $\neg \text{prime } (p - 1)$ **using** $\langle p > 3 \rangle$ *prime-gt-3-impl-p-minus-one-not-prime <prime p>* **by** *auto*
have $p \neq 4$ **using** $\langle \text{prime } p \rangle$ **by** *auto*
hence $p - 1 > 3$ **using** $\langle p > 3 \rangle$ **by** *auto*

then obtain qs where *prod-qs-eq: prod-list qs = p - 1*
and *qs-eq: set qs = prime-factors (p - 1) and qs-length-eq: length qs ≥ 2*
using *prime-factors-list[OF - <¬ prime (p - 1)>]* **by** *auto*
obtain f where $f: \forall q \in \text{prime-factors } (p - 1) . \exists c. f q = c$
 $\wedge ((\text{Prime } q \in \text{set } c) \wedge (\text{valid-cert } c) \wedge \text{length } c \leq 6 * \log 2 q - 4)$
 $\wedge (\forall x \in \text{set } c. \text{size-pratt } x \leq 3 * \log 2 q)$
using *factor-certs by metis*
let $?cs = \text{map } f \text{ } qs$
have $cs: \forall q \in \text{prime-factors } (p - 1) . (\exists c \in \text{set } ?cs . (\text{Prime } q \in \text{set } c) \wedge (\text{valid-cert } c) \wedge \text{length } c \leq 6 * \log 2 q - 4 \wedge (\forall x \in \text{set } c. \text{size-pratt } x \leq 3 * \log 2 q))$
using *f qs-eq by auto*

have *cs-cert-size: $\forall c \in \text{set } ?cs . \forall x \in \text{set } c. \text{size-pratt } x \leq 3 * \log 2 p$*
proof
fix c assume $c \in \text{set } (\text{map } f \text{ } qs)$
then obtain q where $c = f q$ **and** $q \in \text{set } qs$ **by** *auto*
hence $*$: $\forall x \in \text{set } c. \text{size-pratt } x \leq 3 * \log 2 q$ **using** *f qs-eq by blast*
have $q < p \ q > 0$ **using** $qlp \ \langle q \in \text{set } qs \rangle \ \text{qs-eq prime-factors-gt-0-nat}$ **by** *auto*
show $\forall x \in \text{set } c. \text{size-pratt } x \leq 3 * \log 2 p$
proof
fix x assume $x \in \text{set } c$
hence $\text{size-pratt } x \leq 3 * \log 2 q$ **using** $*$ **by** *fastforce*
also have $\dots \leq 3 * \log 2 p$ **using** $\langle q < p \rangle \ \langle q > 0 \rangle \ \langle p > 3 \rangle$ **by** *simp*
finally show $\text{size-pratt } x \leq 3 * \log 2 p$.
qed
qed

have *cs-valid-all: $\forall c \in \text{set } ?cs . \text{valid-cert } c$*
using *f qs-eq by fastforce*

have $\forall x \in \text{set } (\text{build-fpc } p \ a \ (p - 1) \ qs). \text{size-pratt } x \leq 3 * \log 2 p$
using *cs-cert-size a-size <p > 3> prod-qs-eq by (intro size-pratt-fpc) auto*
hence $\forall x \in \text{set } (\text{build-fpc } p \ a \ (p - 1) \ qs @ \text{concat } ?cs) . \text{size-pratt } x \leq 3 * \log 2 p$
using *cs-cert-size by auto*
moreover

```

have Triple p a (p - 1) ∈ set (build-fpc p a (p - 1) qs @ concat ?cs) by (cases
qs) auto
moreover
have valid-cert ((build-fpc p a (p - 1) qs)@ concat ?cs)
proof (rule correct-fpc)
  show valid-cert (concat ?cs)
    using cs-valid-all by (auto simp: valid-cert-concatI)
  show prod-list qs = p - 1 by (rule prod-qs-eq)
  show p - 1 ≠ 0 using prime-gt-1-nat[OF ‹prime p›] by arith
  show ∀ q ∈ set qs . Prime q ∈ set (concat ?cs)
    using concat-set[of prime-factors (p - 1)] cs qs-eq by blast
  show ∀ q ∈ set qs . [a^(p - 1) div q] ≠ 1 (mod p) using qs-eq a by auto
qed (insert ‹p > 3›, simp-all)
moreover
{ let ?k = length qs

  have qs-ge-2: ∀ q ∈ set qs . q ≥ 2 using qs-eq
  by (auto intro: prime-ge-2-nat)

  have ∀ x ∈ set qs. real (length (f x)) ≤ 6 * log 2 (real x) - 4 using f qs-eq by
blast
  hence length (concat ?cs) ≤ (∑ q ← qs. 6 * log 2 q - 4) using concat-length-le
  by fast
  hence length (Prime p # ((build-fpc p a (p - 1) qs)@ concat ?cs))
    ≤ ((∑ q ← (map real qs). 6 * log 2 q - 4) + ?k + 2)
    by (simp add: o-def length-fpc)
  also have ... = (6 * (∑ q ← (map real qs). log 2 q) + (-4 * real ?k) + ?k +
2)
    by (simp add: o-def sum-list-subtractf sum-list-triv sum-list-const-mult)
  also have ... ≤ 6 * log 2 (p - 1) - 4 using ‹?k ≥ 2› prod-qs-eq sum-list-log[of
2 qs] qs-ge-2
    by force
  also have ... ≤ 6 * log 2 p - 4 using log-le-cancel-iff[of 2 p - 1 p] ‹p > 3›
by force
  ultimately have length (Prime p # ((build-fpc p a (p - 1) qs)@ concat ?cs))
    ≤ 6 * log 2 p - 4 by linarith }
  ultimately obtain c where c: Triple p a (p - 1) ∈ set c valid-cert c
    length (Prime p # c) ≤ 6 * log 2 p - 4
    (∀ x ∈ set c. size-pratt x ≤ 3 * log 2 p) by blast
  hence Prime p ∈ set (Prime p # c) valid-cert (Prime p # c)
    (∀ x ∈ set (Prime p # c). size-pratt x ≤ 3 * log 2 p)
  using a ‹prime p› by (auto simp: Primes.prime-gt-Suc-0-nat)
  thus ?case using c by blast
qed
qed

```

We now recapitulate our results. A number p is prime if and only if there is a certificate for p . Moreover, for a prime p there always is a certificate whose size is polynomially bounded in the logarithm of p .

corollary *pratt*:

prime $p \longleftrightarrow (\exists c. \text{Prime } p \in \text{set } c \wedge \text{valid-cert } c)$

using *pratt-complete'* *pratt-sound*(1) by *blast*

corollary *pratt-size*:

assumes *prime* p

shows $\exists c. \text{Prime } p \in \text{set } c \wedge \text{valid-cert } c \wedge \text{size-cert } c \leq (6 * \log 2 p - 4) * (1 + 3 * \log 2 p)$

proof –

obtain c where $c: \text{Prime } p \in \text{set } c \text{ valid-cert } c$

and *len*: $\text{length } c \leq 6 * \log 2 p - 4$ and $(\forall x \in \text{set } c. \text{size-pratt } x \leq 3 * \log 2 p)$

using *pratt-complete'* *assms* by *blast*

hence $\text{size-cert } c \leq \text{length } c * (1 + 3 * \log 2 p)$ by (*simp add: size-pratt-le*)

also have $\dots \leq (6 * \log 2 p - 4) * (1 + 3 * \log 2 p)$ using *len* by *simp*

finally show *?thesis* using c by *blast*

qed

1.3 Efficient modular exponentiation

locale *efficient-power* =

fixes $f :: 'a \Rightarrow 'a \Rightarrow 'a$

assumes *f-assoc*: $\bigwedge x z. f x (f x z) = f (f x x) z$

begin

function *efficient-power* :: $'a \Rightarrow 'a \Rightarrow \text{nat} \Rightarrow 'a$ **where**

efficient-power $y x 0 = y$

| *efficient-power* $y x (\text{Suc } 0) = f x y$

| $n \neq 0 \implies \text{even } n \implies \text{efficient-power } y x n = \text{efficient-power } y (f x x) (n \text{ div } 2)$

| $n \neq 1 \implies \text{odd } n \implies \text{efficient-power } y x n = \text{efficient-power } (f x y) (f x x) (n \text{ div } 2)$

by *force+*

termination by (*relation measure* ($\text{snd} \circ \text{snd}$)) (*auto elim: oddE*)

lemma *efficient-power-code*:

efficient-power $y x n =$

(if $n = 0$ then y

else if $n = 1$ then $f x y$

else if even n then *efficient-power* $y (f x x) (n \text{ div } 2)$

else *efficient-power* $(f x y) (f x x) (n \text{ div } 2)$)

by (*induction y x n rule: efficient-power.induct*) *auto*

lemma *efficient-power-correct*: *efficient-power* $y x n = (f x \overset{\sim}{\sim} n) y$

proof –

have [*simp*]: $f \overset{\sim}{\sim} 2 = (\lambda x. f (f x))$ for $f :: 'a \Rightarrow 'a$

by (*simp add: eval-nat-numeral o-def*)

show *?thesis*

by (*induction y x n rule: efficient-power.induct*)

(*auto elim!: evenE oddE simp: funpow-mult [symmetric] funpow-Suc-right*)

f-assoc *simp del: funpow.simps(2)*

qed

end

interpretation *mod-exp-nat: efficient-power* $\lambda x y :: \text{nat}. (x * y) \text{ mod } m$
by *standard (simp add: mod-mult-left-eq mod-mult-right-eq mult-ac)*

definition *mod-exp-nat-aux* **where** *mod-exp-nat-aux = mod-exp-nat.efficient-power*

lemma *mod-exp-nat-aux-code* [*code*]:

mod-exp-nat-aux m y x n =
(if n = 0 then y
*else if n = 1 then (x * y) mod m*
*else if even n then mod-exp-nat-aux m y ((x * x) mod m) (n div 2)*
*else mod-exp-nat-aux m ((x * y) mod m) ((x * x) mod m) (n div 2))*
unfolding *mod-exp-nat-aux-def* **by** (*rule mod-exp-nat.efficient-power-code*)

lemma *mod-exp-nat-aux-correct*:

*mod-exp-nat-aux m y x n mod m = (x ^ n * y) mod m*

proof –

have *mod-exp-nat-aux m y x n = ((λy. x * y mod m) ^ n) y*
by (*simp add: mod-exp-nat-aux-def mod-exp-nat.efficient-power-correct*)
also have *((λy. x * y mod m) ^ n) y mod m = (x ^ n * y) mod m*

proof (*induction n*)

case (*Suc n*)

hence *x * ((λy. x * y mod m) ^ n) y mod m = x * x ^ n * y mod m*
by (*metis mod-mult-right-eq mult.assoc*)

thus *?case* **by** *auto*

qed *auto*

finally show *?thesis* .

qed

definition *mod-exp-nat* $:: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$

where [*code-abbrev*]: *mod-exp-nat b e m = (b ^ e) mod m*

lemma *mod-exp-nat-code* [*code*]: *mod-exp-nat b e m = mod-exp-nat-aux m 1 b e mod m*

by (*simp add: mod-exp-nat-def mod-exp-nat-aux-correct*)

lemmas [*code-unfold*] = *cong-def*

lemma *eval-mod-exp-nat-aux* [*simp*]:

mod-exp-nat-aux m y x 0 = y
*mod-exp-nat-aux m y x (Suc 0) = (x * y) mod m*
mod-exp-nat-aux m y x (numeral (num.Bit0 n)) =
mod-exp-nat-aux m y (x² mod m) (numeral n)
mod-exp-nat-aux m y x (numeral (num.Bit1 n)) =

```

    mod-exp-nat-aux m ((x * y) mod m) (x2 mod m) (numeral n)
proof –
  define n' where n' = (numeral n :: nat)
  have [simp]: n' ≠ 0 by (auto simp: n'-def)

  show mod-exp-nat-aux m y x 0 = y and mod-exp-nat-aux m y x (Suc 0) = (x *
y) mod m
    by (simp-all add: mod-exp-nat-aux-def)

  have numeral (num.Bit0 n) = (2 * n^)
    by (subst numeral.numeral-Bit0) (simp del: arith-simps add: n'-def)
  also have mod-exp-nat-aux m y x ... = mod-exp-nat-aux m y (x2 mod m) n'
    by (subst mod-exp-nat-aux-code) (simp-all add: power2-eq-square)
  finally show mod-exp-nat-aux m y x (numeral (num.Bit0 n)) =
    mod-exp-nat-aux m y (x2 mod m) (numeral n)
    by (simp add: n'-def)

  have numeral (num.Bit1 n) = Suc (2 * n^)
    by (subst numeral.numeral-Bit1) (simp del: arith-simps add: n'-def)
  also have mod-exp-nat-aux m y x ... = mod-exp-nat-aux m ((x * y) mod m)
(x2 mod m) n'
    by (subst mod-exp-nat-aux-code) (simp-all add: power2-eq-square)
  finally show mod-exp-nat-aux m y x (numeral (num.Bit1 n)) =
    mod-exp-nat-aux m ((x * y) mod m) (x2 mod m) (numeral n)
    by (simp add: n'-def)
qed

lemma eval-mod-exp [simp]:
  mod-exp-nat b' 0 m' = 1 mod m'
  mod-exp-nat b' 1 m' = b' mod m'
  mod-exp-nat b' (Suc 0) m' = b' mod m'
  mod-exp-nat b' e' 0 = b' ^ e'
  mod-exp-nat b' e' 1 = 0
  mod-exp-nat b' e' (Suc 0) = 0
  mod-exp-nat 0 1 m' = 0
  mod-exp-nat 0 (Suc 0) m' = 0
  mod-exp-nat 0 (numeral e) m' = 0
  mod-exp-nat 1 e' m' = 1 mod m'
  mod-exp-nat (Suc 0) e' m' = 1 mod m'
  mod-exp-nat (numeral b) (numeral e) (numeral m) =
    mod-exp-nat-aux (numeral m) 1 (numeral b) (numeral e) mod numeral m
  by (simp-all add: mod-exp-nat-def mod-exp-nat-aux-correct)

```

1.4 Executable certificate checker

```

lemmas [code] = valid-cert.simps(1)

```

```

context
begin

```

lemma *valid-cert-Cons1* [code]:
valid-cert (*Prime p # xs*) \longleftrightarrow
 $p > 1 \wedge (\exists t \in \text{set } xs. \text{case } t \text{ of } \text{Prime} - \Rightarrow \text{False} \mid$
 $\text{Triple } p' a x \Rightarrow p' = p \wedge x = p - 1 \wedge \text{mod-exp-nat } a (p-1) p = 1) \wedge$
valid-cert xs
(is ?lhs = ?rhs)

proof
assume ?lhs **thus** ?rhs **by** (*auto simp: mod-exp-nat-def cong-def split: Pratt.splits*)
next
assume ?rhs
hence $p > 1$ *valid-cert xs* **by** *blast+*
moreover from <?rhs> **obtain** *t* **where** $t \in \text{set } xs$ *case t of Prime - \Rightarrow False* |
 $\text{Triple } p' a x \Rightarrow p' = p \wedge x = p - 1 \wedge [a^{p-1} = 1] \pmod{p}$
by (*auto simp: cong-def mod-exp-nat-def cong: Pratt.case-cong*)
ultimately show ?lhs **by** (*cases t auto*)
qed

private lemma *Suc-0-mod-eq-Suc-0-iff*:
 $\text{Suc } 0 \text{ mod } n = \text{Suc } 0 \longleftrightarrow n \neq \text{Suc } 0$
proof –
consider $n = 0 \mid n = \text{Suc } 0 \mid n > 1$ **by** (*cases n auto*)
thus ?thesis **by** *cases auto*
qed

private lemma *Suc-0-eq-Suc-0-mod-iff*:
 $\text{Suc } 0 = \text{Suc } 0 \text{ mod } n \longleftrightarrow n \neq \text{Suc } 0$
using *Suc-0-mod-eq-Suc-0-iff* **by** (*simp add: eq-commute*)

lemma *valid-cert-Cons2* [code]:
valid-cert (*Triple p a x # xs*) $\longleftrightarrow x > 0 \wedge p > 1 \wedge (x = 1 \vee ($
 $(\exists t \in \text{set } xs. \text{case } t \text{ of } \text{Prime} - \Rightarrow \text{False} \mid$
 $\text{Triple } p' a' y \Rightarrow p' = p \wedge a' = a \wedge y \text{ dvd } x \wedge$
 $(\text{let } q = x \text{ div } y \text{ in } \text{Prime } q \in \text{set } xs \wedge \text{mod-exp-nat } a ((p-1) \text{ div } q) p \neq$
 $1)))) \wedge \text{valid-cert } xs$
(is ?lhs = ?rhs)

proof
assume ?lhs
from <?lhs> **have** *pos: x > 0 and gt-1: p > 1 and valid: valid-cert xs* **by** *simp-all*
show ?rhs
proof (*cases x = 1*)
case *True*
with <?lhs> **show** ?thesis **by** *auto*
next
case *False*
with <?lhs> **have** $(\exists q y. x = q * y \wedge \text{Prime } q \in \text{set } xs \wedge \text{Triple } p a y \in \text{set } xs$
 $\wedge [a^{(p-1) \text{ div } q} \neq 1] \pmod{p})$ **by** *auto*
then obtain *q y* **where** *qy*:
 $x = q * y$

```

    Prime q ∈ set xs
    Triple p a y ∈ set xs
    [a ^ ((p - 1) div q) ≠ 1] (mod p)
  by blast
  hence (∃ t ∈ set xs. case t of Prime - ⇒ False |
    Triple p' a' y ⇒ p' = p ∧ a' = a ∧ y dvd x ∧
    (let q = x div y in Prime q ∈ set xs ∧ mod-exp-nat a ((p-1) div q) p ≠ 1))
  using pos gt-1 by (intro bexI [of - Triple p a y])
  (auto simp: Suc-0-mod-eq-Suc-0-iff Suc-0-eq-Suc-0-mod-iff cong-def mod-exp-nat-def)
  with pos gt-1 valid show ?thesis by blast
qed
next
assume ?rhs
hence pos: x > 0 and gt-1: p > 1 and valid: valid-cert xs by simp-all
show ?lhs
proof (cases x = 1)
  case True
  with ‹?rhs› show ?thesis by auto
next
  case False
  with ‹?rhs› obtain t where t: t ∈ set xs case t of Prime x ⇒ False
    | Triple p' a' y ⇒ p' = p ∧ a' = a ∧ y dvd x ∧ (let q = x div y
    in Prime q ∈ set xs ∧ mod-exp-nat a ((p - 1) div q) p ≠ 1) by auto
  then obtain y where y: t = Triple p a y y dvd x let q = x div y in Prime q
  ∈ set xs ∧
    mod-exp-nat a ((p - 1) div q) p ≠ 1
  by (cases t rule: pratt.exhaust) auto
  with gt-1 have y': let q = x div y in Prime q ∈ set xs ∧ [a ^ ((p - 1) div q) ≠
  1] (mod p)
  by (auto simp: cong-def Let-def mod-exp-nat-def Suc-0-mod-eq-Suc-0-iff Suc-0-eq-Suc-0-mod-iff)
  define q where q = x div y
  have ∃ q y. x = q * y ∧ Prime q ∈ set xs ∧ Triple p a y ∈ set xs
    ∧ [a ^ ((p - 1) div q) ≠ 1] (mod p)
  by (rule exI[of - q], rule exI[of - y]) (insert t y y', auto simp: Let-def q-def)
  with pos gt-1 valid show ?thesis by simp
qed
qed
declare valid-cert.simps(2,3) [simp del]
lemmas eval-valid-cert = valid-cert.simps(1) valid-cert-Cons1 valid-cert-Cons2
end

```

The following alternative tree representation of certificates is better suited for efficient checking.

```

datatype pratt-tree = Pratt-Node nat × nat × pratt-tree list

fun pratt-tree-number where

```


pratt-tree-number (*Pratt-Node* (*n*, -, -)) = *n*

The following function checks that a given list contains all the prime factors of the given number.

fun *check-prime-factors-subset* :: *nat* \Rightarrow *nat list* \Rightarrow *bool* **where**
check-prime-factors-subset *n* [] \longleftrightarrow *n* = 1
| *check-prime-factors-subset* *n* (*p* # *ps*) \longleftrightarrow (if *n* = 0 then *False* else
(if *p* > 1 \wedge *p* *dvd* *n* then *check-prime-factors-subset* (*n* *div* *p*) (*p* # *ps*)
else *check-prime-factors-subset* *n* *ps*))

lemma *check-prime-factors-subset-0* [*simp*]: \neg *check-prime-factors-subset* 0 *ps*
by (*induction ps*) *auto*

lemmas [*simp del*] = *check-prime-factors-subset.simps*(2)

lemma *check-prime-factors-subset-Cons* [*simp*]:
check-prime-factors-subset (*Suc* 0) (*p* # *ps*) \longleftrightarrow *check-prime-factors-subset* (*Suc*
0) *ps*
check-prime-factors-subset 1 (*p* # *ps*) \longleftrightarrow *check-prime-factors-subset* 1 *ps*
p > 1 \implies *p* *dvd* *numeral n* \implies *check-prime-factors-subset* (*numeral n*) (*p* # *ps*)
 \longleftrightarrow
check-prime-factors-subset (*numeral n* *div* *p*) (*p* # *ps*)
p \leq 1 \vee \neg *p* *dvd* *numeral n* \implies *check-prime-factors-subset* (*numeral n*) (*p* # *ps*)
 \longleftrightarrow
check-prime-factors-subset (*numeral n*) *ps*
by (*subst check-prime-factors-subset.simps; force*)+

lemma *check-prime-factors-subset-correct*:
assumes *check-prime-factors-subset n ps list-all prime ps*
shows *prime-factors n* \subseteq *set ps*
using *assms*
proof (*induction n ps rule: check-prime-factors-subset.induct*)
case (2 *n p ps*)
note * = *this*
from 2.prem1 **have** *prime p* **and** *p* > 1
by (*auto simp: prime-gt-Suc-0-nat*)

consider *n* = 0 | *n* > 0 *p* *dvd* *n* | *n* > 0 \neg (*p* *dvd* *n*)
by *blast*
thus ?*case*
proof *cases*
case 2
hence *n* *div* *p* > 0 **by** *auto*
hence *prime-factors* ((*n* *div* *p*) * *p*) = *insert p* (*prime-factors* (*n* *div* *p*))
using <*p* > 1> <*prime p*> **by** (*auto simp: prime-factors-product prime-prime-factors*)
also have (*n* *div* *p*) * *p* = *n*
using 2 **by** *auto*
finally show ?*thesis* **using** 2 <*p* > 1> *
by (*auto simp: check-prime-factors-subset.simps*(2)[*of n*])

```

next
  case 3
  with * and ⟨p > 1⟩ show ?thesis
  by (auto simp: check-prime-factors-subset.simps(2)[of n])
qed auto
qed auto

```

```

fun valid-pratt-tree where
  valid-pratt-tree (Pratt-Node (n, a, ts)) ⟷
    n ≥ 2 ∧
    check-prime-factors-subset (n - 1) (map pratt-tree-number ts) ∧
    [a ^ (n - 1) = 1] (mod n) ∧
    (∀ t ∈ set ts. [a ^ ((n - 1) div pratt-tree-number t) ≠ 1] (mod n)) ∧
    (∀ t ∈ set ts. valid-pratt-tree t)

```

```

lemma valid-pratt-tree-code [code]:
  valid-pratt-tree (Pratt-Node (n, a, ts)) ⟷
    n ≥ 2 ∧
    check-prime-factors-subset (n - 1) (map pratt-tree-number ts) ∧
    mod-exp-nat a (n - 1) n = 1 ∧
    (∀ t ∈ set ts. mod-exp-nat a ((n - 1) div pratt-tree-number t) n ≠ 1) ∧
    (∀ t ∈ set ts. valid-pratt-tree t)
  by (simp add: mod-exp-nat-def cong-def)

```

```

lemma valid-pratt-tree-imp-prime:
  assumes valid-pratt-tree t
  shows prime (pratt-tree-number t)
  using assms
proof (induction t rule: valid-pratt-tree.induct)
  case (1 n a ts)
  from 1 have prime-factors (n - 1) ⊆ set (map pratt-tree-number ts)
  by (intro check-prime-factors-subset-correct) (auto simp: list.pred-set)
  with 1 show ?case
  by (intro lehmers-theorem[where a = a]) auto
qed

```

```

lemma valid-pratt-tree-imp-prime':
  assumes PROP (Trueprop (valid-pratt-tree (Pratt-Node (n, a, ts)))) ≡ PROP
  (Trueprop True)
  shows prime n
proof -
  have valid-pratt-tree (Pratt-Node (n, a, ts))
  by (subst assms) auto
  from valid-pratt-tree-imp-prime[OF this] show ?thesis by simp
qed

```

1.5 Proof method setup

theorem *lehmers-theorem'*:

```

fixes p :: nat
assumes list-all prime ps a  $\equiv$  a n  $\equiv$  n
assumes list-all ( $\lambda p.$  mod-exp-nat a ((n - 1) div p) n  $\neq$  1) ps mod-exp-nat a
(n - 1) n = 1
assumes check-prime-factors-subset (n - 1) ps 2  $\leq$  n
shows prime n
using assms check-prime-factors-subset-correct[OF assms(6,1)]
by (intro lehmers-theorem[where a = a]) (auto simp: cong-def mod-exp-nat-def
list.pred-set)

```

lemma *list-all-ConsI*: $P\ x \implies \text{list-all } P\ xs \implies \text{list-all } P\ (x \# xs)$

by *simp*

ML-file \langle *pratt.ML* \rangle

method-setup *pratt* = \langle

```

Scan.lift (Pratt.tac-config-parser -- Scan.option Pratt.cert-cartouche) >>
(fn (config, cert) => fn ctxt => SIMPLE-METHOD (HEADGOAL (Pratt.tac
config cert ctxt)))
 $\rangle$  Prove primality of natural numbers using Pratt certificates.

```

The proof method replays a given Pratt certificate to prove the primality of a given number. If no certificate is given, the method attempts to compute one. The computed certificate is then also printed with a prompt to insert it into the proof document so that it does not have to be recomputed the next time.

The format of the certificates is compatible with those generated by Mathematica. Therefore, for larger numbers, certificates generated by Mathematica can be used with this method directly.

lemma *prime (47 :: nat)*

by (*pratt (silent)*)

lemma *prime (2503 :: nat)*

by *pratt*

lemma *prime (7919 :: nat)*

by *pratt*

lemma *prime (131059 :: nat)*

by (*pratt* \langle {131059, 2, {2, {3, 2, {2}}, {809, 3, {2, {101, 2, {2, {5, 2, {2}}}}}}}} \rangle)

end

theory *Pratt-Certificate-Code*

imports

Pratt-Certificate
HOL-Library.Code-Target-Numeral
begin

1.6 Code generation for Pratt certificates

The following one-time setup is required to set up code generation for the certificate checking. Other theories importing this theories do not have to do this again.

```
setup <
  Context.theory-map (Pratt.setup-valid-cert-code-conv
    (@{computation-check
      terms: Trueprop valid-pratt-tree 0::nat 1::nat 2::nat 3::nat 4::nat
      datatypes: pratt-tree list nat × nat × pratt-tree list nat}))
  >
```

We can now evaluate the efficiency of the procedure on some examples.

```
lemma prime (131059 :: nat)
  by (pratt (code))
```

```
lemma prime (100000007 :: nat)
  by (pratt (code))
```

```
lemma prime (8504276003 :: nat)
  by (pratt (code))
```

```
lemma prime (52759926861157 :: nat)
  by (pratt (code))
```

```
lemma prime (39070009756439177203 :: nat)
  by (pratt (code)
    <{39070009756439177203, 2,
      {2, {3, 2, {2}}, {197, 2, {2, {7, 3, {2, {3, 2, {2}}}}}},
      {11018051256751037, 2, {2, {19, 2, {2, {3, 2, {2}}}}},
      {1249, 7, {2, {3, 2, {2}}, {13, 2, {2, {3, 2, {2}}}}},
      {116072344789, 2, {2, {3, 2, {2}}},
      {3067, 2, {2, {3, 2, {2}}, {7, 3, {2, {3, 2, {2}}}}},
      {73, 5, {2, {3, 2, {2}}}}},
      {3153797, 2, {2, {788449, 11,
      {2, {3, 2, {2}}, {43, 3,
      {2, {3, 2, {2}}, {7, 3, {2, {3, 2, {2}}}}}},
      {191, 19, {2, {5, 2, {2}}, {19, 2, {2, {3, 2, {2}}}}}}}}}}>
```

```
lemma prime (933491553728809239092563013853810654040043466297416456476877
:: nat)
  by (pratt (code)
    <{933491553728809239092563013853810654040043466297416456476877,
      2, {2, {38463351105299604725411, 6,
```

```

{2, {5, 2, {2}}, {13, 2, {2, {3, 2, {2}}}},
{295871931579227728657, 5,
{2, {3, 2, {2}}, {26041, 13,
{2, {3, 2, {2}}, {5, 2, {2}}, {7, 3, {2, {3, 2, {2}}}},
{31, 3, {2, {3, 2, {2}}, {5, 2, {2}}}}}},
{29009, 3, {2, {7, 3, {2, {3, 2, {2}}}},
{37, 2, {2, {3, 2, {2}}}}}},
{39133, 5, {2, {3, 2, {2}},
{1087, 3, {2, {3, 2, {2}},
{181, 2, {2, {3, 2, {2}}, {5, 2, {2}}}}}}}},
{208511, 7, {2, {5, 2, {2}}, {29, 2, {2, {7, 3, {2, {3, 2, {2}}}}}},

{719, 11, {2, {359, 7,
{2, {179, 2, {2, {89, 3, {2, {11, 2, {2, {5, 2, {2}}}}}}}}}}}}}}}}}}}}}}
}}}}, {6067409149902371339956289140415169329, 3,
{2, {11, 2, {2, {5, 2, {2}}}}},
{103, 5, {2, {3, 2, {2}}, {17, 3, {2}}}},
{7955023343, 7, {2, {7, 3, {2, {3, 2, {2}}}},
{79, 3, {2, {3, 2, {2}}, {13, 2, {2, {3, 2, {2}}}}}}}},
{1451, 2, {2, {5, 2, {2}}, {29, 2, {2, {7, 3, {2, {3, 2, {2}}}}}}}}}},

{4957, 2, {2, {3, 2, {2}}, {7, 3, {2, {3, 2, {2}}}},
{59, 2, {2, {29, 2, {2, {7, 3, {2, {3, 2, {2}}}}}}}}}}}}},
{8108847767, 5, {2, {4054423883, 2,
{2, {2027211941, 2, {2, {5, 2, {2}}, {13, 2, {2, {3, 2, {2}}}},
{29, 2, {2, {7, 3, {2, {3, 2, {2}}}}}}}},
{268861, 6, {2, {3, 2, {2}}, {5, 2, {2}},
{4481, 3, {2, {5, 2, {2}}, {7, 3, {2, {3, 2, {2}}}}}}}}}}}}}}}}}}}},

{5188630976471, 13, {2, {5, 2, {2}},
{518863097647, 5, {2, {3, 2, {2}},
{67, 2, {2, {3, 2, {2}}, {11, 2, {2, {5, 2, {2}}}}}}}},
{17881, 7, {2, {3, 2, {2}}, {5, 2, {2}},
{149, 2, {2, {37, 2, {2, {3, 2, {2}}}}}}}}}},
{24061, 10, {2, {3, 2, {2}}, {5, 2, {2}},
{401, 3, {2, {5, 2, {2}}}}}}}}}}}}}}}}}}}}

```

)

end

References

- [1] V. R. Pratt. Every prime has a succinct certificate. *SIAM Journal on Computing*, 4(3):214–220, 1975.