A Formalization of Pratt's Primality Certificates

By Simon Wimmer and Lars Noschinski

December 7, 2022

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 . . . 3
- 1.3 Efficient modular exponentiation . . . 6
- 1.4 Executable certificate checker . . . 8
- 1.5 Proof method setup . . . 10
- 1.6 Code generation for Pratt certificates . . . 11

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:

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

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 $[a\hat{}((p - 1)\ div\ q) \neq 1]\ (mod\ p)$ hold for some prime number q we can conclude (p, a, qx) from that.
- R2: If we know that $(p, a, p - 1)$ and $[a\hat{}(p - 1) = 1]\ (mod\ p)$ hold, we can infer 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 ⇒ bool where
  valid-cert [] = True
| R2: valid-cert (Prime p#xs) ⟷ 1 < p ∧ valid-cert xs
    ∧ (∃ a . [a^(p − 1) = 1] (mod p) ∧ Triple p a (p − 1) ∈ set xs)
| R1: valid-cert (Triple p a x # xs) ⟷ p > 1 ∧ 0 < x ∧ valid-cert xs ∧ (x=1 ∨
    (∃ q y. x = q * y ∧ Prime q ∈ set xs ∧ Triple p a y ∈ set xs
      ∧ [a^((p − 1) div q) ≠ 1] (mod 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 ⇒ 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 ⇒ real where
  size-cert [] = 0 |
  size-cert (x # xs) = 1 + size-pratt x + size-cert xs
```

1.1 Soundness

In Section 1 we introduced the predicates 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 Prime(p) occurs in a certificate, p is prime.

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

⟨*proof*⟩

lemma *prime-factors-of-prime*: **fixes** p :: *nat* **assumes** *prime p* **shows** *prime-factors* $p = \{p\}$
⟨*proof*⟩

definition *pratt-triple* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
pratt-triple p a x ⟷ $x > 0 \wedge (\forall q \in$ *prime-factors x*. $[a \hat{} ((p - 1)\ div\ q) \neq 1]$ $(mod\ p))$

lemma *pratt-triple-1*: $p > 1 \Longrightarrow x = 1 \Longrightarrow$ *pratt-triple p a x*
⟨*proof*⟩

lemma *pratt-triple-extend*:
assumes *prime q pratt-triple p a y*
$p > 1\ x > 0\ x = q * y\ [a \hat{} ((p - 1)\ div\ q) \neq 1]\ (mod\ p)$
shows *pratt-triple p a x*
⟨*proof*⟩

lemma *pratt-triple-imp-prime*:
assumes *pratt-triple p a x* $p > 1\ x = p - 1\ [a \hat{} (p - 1) = 1]\ (mod\ p)$
shows *prime p*
⟨*proof*⟩

theorem *pratt-sound*:
assumes *1*: *valid-cert c*
assumes *2*: $t \in$ *set c*
shows $(t =$ *Prime p* $\longrightarrow$ *prime p*$) \wedge$
$(t =$ *Triple p a x* $\longrightarrow ((\forall q \in$ *prime-factors x* . $[a \hat{} ((p - 1)\ div\ q) \neq 1]$ $(mod\ p)) \wedge 0<x))$
⟨*proof*⟩

corollary *pratt-primeI*:
assumes *valid-cert xs Prime p* $\in$ *set xs*
shows *prime p*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *valid-cert-concatI*: (∀ *x* ∈ *set xs* . *valid-cert x*) ⟹ *valid-cert* (*concat xs*)
⟨*proof*⟩

lemma *size-pratt-le*:
fixes *d*::*real*
assumes ∀ *x* ∈ *set c*. *size-pratt x* ≤ *d*
shows *size-cert c* ≤ *length c* ∗ (*1* + *d*) ⟨*proof*⟩

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 $q_1 \cdot \dots \cdot q_n = p - 1$, 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 and a certificate rs for all prime factors of p, then we can construct a certificate for p as

Prime p # *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 ∀ *q* ∈ *set qs* . *Prime q* ∈ *set xs*
assumes ∀ *q* ∈ *set qs* . [*a*^((*p* − *1*) *div q*) ≠ *1*] (*mod p*)
shows *valid-cert* (*build-fpc p a r qs* @ *xs*)
⟨*proof*⟩

lemma *length-fpc*:
length (*build-fpc p a r qs*) = *length qs* + *1* ⟨*proof*⟩

lemma *div-gt-0*:
fixes *m n* :: *nat* **assumes** *m* ≤ *n 0* < *m* **shows** *0* < *n div m*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *concat-set*:
assumes $\forall\ q \in qs\ .\ \exists\ c \in set\ cs\ .\ Prime\ q \in set\ c$
shows $\forall\ q \in qs\ .\ Prime\ q \in set\ (concat\ cs)$
⟨*proof*⟩

lemma *p-in-prime-factorsE*:
fixes $n :: nat$
assumes $p \in prime\text{-}factors\ n\ 0 < n$
obtains $2 \leq p\ p \leq n\ p\ dvd\ n\ prime\ p$
⟨*proof*⟩

lemma *prime-factors-list-prime*:
fixes $n :: nat$
assumes *prime n*
shows $\exists\ qs.\ prime\text{-}factors\ n = set\ qs \wedge prod\text{-}list\ qs = n \wedge length\ qs = 1$
⟨*proof*⟩

lemma *prime-factors-list*:
fixes $n :: nat$ **assumes** $3 < n \neg\ prime\ n$
shows $\exists\ qs.\ prime\text{-}factors\ n = set\ qs \wedge prod\text{-}list\ qs = n \wedge length\ qs \geq 2$
⟨*proof*⟩

lemma *prod-list-ge*:
fixes *xs*::*nat list*
assumes $\forall\ x \in set\ xs\ .\ x \geq 1$
shows $prod\text{-}list\ xs \geq 1$ ⟨*proof*⟩

lemma *sum-list-log*:
fixes *b*::*real*
fixes *xs*::*nat list*
assumes *b*: $b > 0\ b \neq 1$
assumes *xs*:$\forall\ x \in set\ xs\ .\ x \geq b$
shows $(\sum x{\leftarrow}xs.\ log\ b\ x) = log\ b\ (prod\text{-}list\ xs)$
⟨*proof*⟩

lemma *concat-length-le*:
fixes $g :: nat \Rightarrow real$
assumes $\forall\ x \in set\ xs\ .\ real\ (length\ (f\ x)) \leq g\ x$
shows $length\ (concat\ (map\ f\ xs)) \leq (\sum x{\leftarrow}xs.\ g\ x)$ ⟨*proof*⟩

lemma *prime-gt-3-impl-p-minus-one-not-prime*:
fixes *p*::*nat*
assumes *prime p p*>*3*
shows $\neg\ prime\ (p - 1)$
⟨*proof*⟩

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 $\exists\, c.$ *Prime p* $\in$ *set c* $\wedge$ *valid-cert c* $\wedge$ *length c* $\leq$ *6∗log 2 p − 4* $\wedge$ ($\forall$ *x* $\in$ *set c. size-pratt x* $\leq$ *3 ∗ log 2 p*) ⟨*proof*⟩

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 $\longleftrightarrow$ ($\exists\, c.$ *Prime p* $\in$ *set c* $\wedge$ *valid-cert c*)
⟨*proof*⟩

corollary *pratt-size*:
assumes *prime p*
shows $\exists\, c.$ *Prime p* $\in$ *set c* $\wedge$ *valid-cert c* $\wedge$ *size-cert c* $\leq$ (*6 ∗ log 2 p − 4*) ∗ (*1 + 3 ∗ log 2 p*)
⟨*proof*⟩

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* $\Rightarrow$ *nat* $\Rightarrow$ *′a* **where**
efficient-power y x 0 = *y*
| *efficient-power y x* (*Suc 0*) = *f x y*
| *n* $\neq$ *0* $\Longrightarrow$ *even n* $\Longrightarrow$ *efficient-power y x n* = *efficient-power y* (*f x x*) (*n div 2*)
| *n* $\neq$ *1* $\Longrightarrow$ *odd n* $\Longrightarrow$ *efficient-power y x n* = *efficient-power* (*f x y*) (*f x x*) (*n div 2*)
⟨*proof*⟩
termination ⟨*proof*⟩

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 div 2*)
else efficient-power (*f x y*) (*f x x*) (*n div 2*))
⟨*proof*⟩

lemma *efficient-power-correct*: *efficient-power y x n* = (*f x* $\frown$ *n*) *y*
⟨*proof*⟩

end

interpretation *mod-exp-nat*: *efficient-power* λ*x y* :: *nat*. (*x ∗ y*) *mod m*
⟨*proof*⟩

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))*
⟨*proof*⟩

lemma *mod-exp-nat-aux-correct*:
*mod-exp-nat-aux m y x n mod m = (x ^ n * y) mod m*
⟨*proof*⟩

definition *mod-exp-nat* :: *nat ⇒ nat ⇒ nat ⇒ 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*
⟨*proof*⟩

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 (x^2 mod m) (numeral n)
mod-exp-nat-aux m y x (numeral (num.Bit1 n)) =
*mod-exp-nat-aux m ((x * y) mod m) (x^2 mod m) (numeral n)*
⟨*proof*⟩

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
⟨*proof*⟩

1.4 Executable certificate checker

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

context
begin

lemma *valid-cert-Cons1* [*code*]:
  valid-cert (*Prime p # xs*) ⟷
    p > *1* ∧ (∃ *t*∈*set xs*. *case t of Prime - ⇒ False* |
      Triple p′ a x ⇒ p′ = *p* ∧ *x* = *p* − *1* ∧ *mod-exp-nat a* (*p*−*1*) *p* = *1*) ∧
valid-cert xs
  (**is** *?lhs* = *?rhs*)
⟨*proof*⟩ **lemma** *Suc-0-mod-eq-Suc-0-iff*:
  Suc 0 mod n = *Suc 0* ⟷ *n* ≠ *Suc 0*
⟨*proof*⟩ **lemma** *Suc-0-eq-Suc-0-mod-iff*:
  Suc 0 = *Suc 0 mod n* ⟷ *n* ≠ *Suc 0*
  ⟨*proof*⟩

lemma *valid-cert-Cons2* [*code*]:
  valid-cert (*Triple p a x # xs*) ⟷ *x* > *0* ∧ *p* > *1* ∧ (*x* = *1* ∨ (
    (∃ *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)))) ∧ *valid-cert xs*
  (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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* ⇒ *nat list* ⇒ *bool* **where**
  check-prime-factors-subset n [] ⟷ *n* = *1*
| *check-prime-factors-subset n* (*p # ps*) ⟷ (*if n* = *0 then False else*
    (*if p* > *1* ∧ *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*]: ¬*check-prime-factors-subset 0 ps*
⟨*proof*⟩

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

lemma *check-prime-factors-subset-Cons* [*simp*]:
check-prime-factors-subset (*Suc 0*) (*p # ps*) ⟷ *check-prime-factors-subset* (*Suc 0*) *ps*
check-prime-factors-subset 1 (*p # ps*) ⟷ *check-prime-factors-subset 1 ps*
p > 1 ⟹ *p dvd numeral n* ⟹ *check-prime-factors-subset* (*numeral n*) (*p # ps*) ⟷
check-prime-factors-subset (*numeral n div p*) (*p # ps*)
p ≤ 1 ∨ ¬*p dvd numeral n* ⟹ *check-prime-factors-subset* (*numeral n*) (*p # ps*) ⟷
check-prime-factors-subset (*numeral n*) *ps*
⟨*proof*⟩

lemma *check-prime-factors-subset-correct*:
assumes *check-prime-factors-subset n ps list-all prime ps*
shows *prime-factors n* ⊆ *set ps*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *valid-pratt-tree-imp-prime*:
assumes *valid-pratt-tree t*
shows *prime* (*pratt-tree-number t*)
⟨*proof*⟩

lemma *valid-pratt-tree-imp-prime′*:
assumes *PROP* (*Trueprop* (*valid-pratt-tree* (*Pratt-Node* (*n*, *a*, *ts*)))) ≡ *PROP* (*Trueprop True*)
shows *prime n*

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *list-all-ConsI*: $P\ x \Longrightarrow$ *list-all* $P\ xs \Longrightarrow$ *list-all* $P\ (x \# xs)$
⟨*proof*⟩

⟨*ML*⟩

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* $(47 :: nat)$
⟨*proof*⟩

lemma *prime* $(2503 :: nat)$
⟨*proof*⟩

lemma *prime* $(7919 :: nat)$
⟨*proof*⟩

lemma *prime* $(131059 :: nat)$
⟨*proof*⟩

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.

⟨*ML*⟩

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

lemma *prime* (*131059* :: *nat*)
⟨*proof*⟩

lemma *prime* (*100000007* :: *nat*)
⟨*proof*⟩

lemma *prime* (*8504276003* :: *nat*)
⟨*proof*⟩

lemma *prime* (*52759926861157* :: *nat*)
⟨*proof*⟩

lemma *prime* (*390700097564391772O3* :: *nat*)
⟨*proof*⟩

lemma *prime* (*93349155372880923909256301385381065404004346629741645647687 7* :: *nat*)
⟨*proof*⟩

end

References

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