Mersenne primes and the Lucas–Lehmer test

Manuel Eberl

June 7, 2023

Abstract

This article provides formal proofs of basic properties of Mersenne numbers, i. e. numbers of the form $2^n - 1$, and especially of Mersenne primes. In particular, an efficient, verified, and executable version of the Lucas–Lehmer test is developed. This test decides primality for Mersenne numbers in time polynomial in n.

Contents

1 Auxiliary material **2**
- 1.1 Auxiliary number-theoretic material . . . 2
- 1.2 Auxiliary algebraic material . . . 6

2 The Lucas–Lehmer test **9**
- 2.1 General properties of Mersenne numbers and Mersenne primes 10
- 2.2 The Lucas–Lehmer sequence . . . 14
- 2.3 The ring $\mathbb{Z}[\sqrt{3}]$ . . . 15
- 2.4 The ring $(\mathbb{Z}/m\mathbb{Z})[\sqrt{3}]$ . . . 16
- 2.5 $\mathbb{Z}[\sqrt{3}]$ as a subring of $\mathbb{R}$ . . . 21
- 2.6 The canonical homomorphism $\mathbb{Z}[\sqrt{3}] \to (\mathbb{Z}/m\mathbb{Z})[\sqrt{3}]$ . . . 23
- 2.7 Correctness of the Lucas–Lehmer test . . . 25
- 2.8 A first executable version Lucas–Lehmer test . . . 34

3 Efficient code for testing Mersenne primes **35**
- 3.1 Efficient computation of remainders modulo a Mersenne number 36
- 3.2 Efficient code for the Lucas–Lehmer sequence . . . 39
- 3.3 Code for the Lucas–Lehmer test . . . 40
- 3.4 Examples . . . 41

1 Auxiliary material

theory *Lucas-Lehmer-Auxiliary*
imports
HOL−Algebra.Ring
Probabilistic-Prime-Tests.Jacobi-Symbol
begin

1.1 Auxiliary number-theoretic material

lemma *congD*: $[a = b]\ (mod\ n) \Longrightarrow a\ mod\ n = b\ mod\ n$
by (*auto simp*: *cong-def*)

lemma *eval-coprime*:
$(b :: 'a :: euclidean\text{-}semiring\text{-}gcd) \neq 0 \Longrightarrow coprime\ a\ b \longleftrightarrow coprime\ b\ (a\ mod\ b)$
by (*simp add*: *coprime-commute*)

lemma *two-power-odd-mod-12*:
assumes *odd n n* > *1*
shows $[2 \hat{} n = 8]\ (mod\ (12 :: nat))$
using *assms*
proof (*induction n rule*: *less-induct*)
case (*less n*)
show *?case*
proof (*cases n* = *3*)
case *False*
with *less.prems* **have** *n* > *3* **by** (*auto elim*!: *oddE*)
hence $[2 \hat{} (n - 2 + 2) = (8 * 4 :: nat)]\ (mod\ 12)$
unfolding *power-add* **using** *less.prems* **by** (*intro cong-mult less*) *auto*
also have *n* − *2* + *2* = *n*
using ‹*n* > *3*› **by** *simp*
finally show *?thesis* **by** (*simp add*: *cong-def*)
qed *auto*
qed

lemma *Legendre-3-right*:
fixes *p* :: *nat*
assumes *p*: *prime p p* > *3*
shows *p mod 12* ∈ {*1*, *5*, *7*, *11*} **and** *Legendre p 3* = (*if p mod 12* ∈ {*1*, *7*} *then 1 else* −*1*)
proof −
have *coprime p 2* **using** *p prime-nat-not-dvd*[*of p 2*]
by (*intro prime-imp-coprime*) (*auto dest*: *dvd-imp-le*)
moreover have *coprime p 3* **using** *p*
by (*intro prime-imp-coprime*) *auto*
ultimately have *coprime p* (*2* ∗ *2* ∗ *3*)
unfolding *coprime-mult-right-iff* **by** *auto*
hence *coprime 12 p*
by (*simp add*: *coprime-commute*)

```
  hence p mod 12 ∈ {p∈{..11}. coprime 12 p} by auto
  also have {p∈{..11}. coprime 12 p} = {1::nat, 5, 7, 11}
    unfolding atMost-nat-numeral pred-numeral-simps arith-simps
    by (auto simp del: coprime-imp-gcd-eq-1 simp: eval-coprime)
  finally show p mod 12 ∈ {1, 5, 7, 11} by auto
  hence p mod 12 = 1 ∨ p mod 12 = 5 ∨ p mod 12 = 7 ∨ p mod 12 = 11
    by auto
  thus Legendre p 3 = (if p mod 12 ∈ {1, 7} then 1 else −1)
  proof safe
    assume p mod 12 = 1
    have Legendre (int p) 3 = Legendre (int p mod 3) 3
      by (intro Legendre-mod [symmetric]) auto
    also from ‹p mod 12 = 1› have p mod 12 mod 3 = 1 by simp
    hence p mod 3 = 1 by (simp add: mod-mod-cancel)
    hence int p mod 3 = 1 by presburger
    finally have Legendre p 3 = 1 by simp
    thus ?thesis using ‹p mod 12 = 1› by simp
  next
    assume p mod 12 = 5
    have Legendre (int p) 3 = Legendre (int p mod 3) 3
      by (intro Legendre-mod [symmetric]) auto
    also from ‹p mod 12 = 5› have p mod 12 mod 3 = 2 by simp
    hence p mod 3 = 2 by (simp add: mod-mod-cancel)
    hence int p mod 3 = 2 by presburger
    finally have Legendre p 3 = −1 by (simp add: supplement2-Legendre)
    thus ?thesis using ‹p mod 12 = 5› by simp
  next
    assume p mod 12 = 7
    have Legendre (int p) 3 = Legendre (int p mod 3) 3
      by (intro Legendre-mod [symmetric]) auto
    also from ‹p mod 12 = 7› have p mod 12 mod 3 = 1 by simp
    hence p mod 3 = 1 by (simp add: mod-mod-cancel)
    hence int p mod 3 = 1 by presburger
    finally have Legendre p 3 = 1 by simp
    thus ?thesis using ‹p mod 12 = 7› by simp
  next
    assume p mod 12 = 11
    have Legendre (int p) 3 = Legendre (int p mod 3) 3
      by (intro Legendre-mod [symmetric]) auto
    also from ‹p mod 12 = 11› have p mod 12 mod 3 = 2 by simp
    hence p mod 3 = 2 by (simp add: mod-mod-cancel)
    hence int p mod 3 = 2 by presburger
    finally have Legendre p 3 = −1 by (simp add: supplement2-Legendre)
    thus ?thesis using ‹p mod 12 = 11› by simp
  qed
qed

lemma Legendre-3-left:
  fixes p :: nat
```

assumes *p*: *prime p p* > *3*
shows *Legendre 3 p* = (*if p mod 12* ∈ {*1*, *11*} *then 1 else* −*1*)
proof (*cases p mod 12* = *1* ∨ *p mod 12* = *5*)
case *True*
hence *p mod 12 mod 4* = *1* **by** *auto*
hence *even* ((*p* − *Suc 0*) *div 2*)
by (*intro even-mod-4-div-2*) (*auto simp*: *mod-mod-cancel*)
with *Quadratic-Reciprocity*[*of p 3*] *Legendre-3-right*(*2*)[*of p*] *assms True* **show** *?thesis*
by *auto*
next
case *False*
with *Legendre-3-right*(*1*)[*OF assms*] **have** ∗: *p mod 12* = *7* ∨ *p mod 12* = *11* **by** *auto*
hence *p mod 12 mod 4* = *3* **by** *auto*
hence *odd* ((*p* − *Suc 0*) *div 2*)
by (*intro odd-mod-4-div-2*) (*auto simp*: *mod-mod-cancel*)
with *Quadratic-Reciprocity*[*of p 3*] *Legendre-3-right*(*2*)[*of p*] *assms* ∗ **show** *?thesis*
by *fastforce*
qed

lemma *supplement2-Legendre′*:
assumes *prime p p* ≠ *2*
shows *Legendre 2 p* = (*if p mod 8* = *1* ∨ *p mod 8* = *7 then 1 else* −*1*)
proof −
from *assms* **have** *p* > *2*
using *prime-gt-1-int*[*of p*] **by** *auto*
moreover from *this* **and** *assms* **have** *odd p*
by (*auto simp*: *prime-odd-int*)
ultimately show *?thesis*
using *supplement2-Jacobi′*[*of p*] *assms prime-odd-int*[*of p*]
by (*simp add*: *prime-p-Jacobi-eq-Legendre*)
qed

lemma *little-Fermat-nat*:
fixes *a* :: *nat*
assumes *prime p* ¬*p dvd a*
shows [*a* ^ *p* = *a*] (*mod p*)
proof −
have *p* = *Suc* (*p* − *1*)
using *prime-gt-0-nat*[*OF assms*(*1*)] **by** *simp*
also have *p* − *1* = *totient p*
using *assms* **by** (*simp add*: *totient-prime*)
also have *a* ^ (*Suc* . . .) = *a* ∗ *a* ^ *totient p*
by *simp*
also have [. . . = *a* ∗ *1*] (*mod p*)
using *prime-imp-coprime*[*of p a*] *assms*
by (*intro cong-mult cong-refl euler-theorem*) (*auto simp*: *coprime-commute*)
finally show *?thesis* **by** *simp*

qed

lemma *little-Fermat-int*:
fixes $a :: int$ **and** $p :: nat$
assumes *prime p* $\neg p$ *dvd a*
shows $[a \hat{} p = a]$ (*mod p*)
proof −
have $p > 1$ **using** *prime-gt-1-nat assms* **by** *simp*
have $\neg$*int p dvd a mod int p*
using *assms* **by** (*simp add*: *dvd-mod-iff*)
also from ‹$p > 1$› **have** *a mod int p* = *int* (*nat* (*a mod int p*))
by *simp*
finally have *not-dvd*: $\neg p$ *dvd nat* (*a mod int p*)
by (*subst* (*asm*) *int-dvd-int-iff*)

have $[a \hat{} p = (a \text{ mod } p) \hat{} p]$ (*mod p*)
by (*intro cong-pow*) (*auto simp*: *cong-def*)
also have (*a mod p*) $\hat{}$ *p* = (*int* (*nat* (*a mod p*))) $\hat{}$ *p*
using ‹$p > 1$› **by** (*subst of-nat-nat*) *auto*
also have ... = *int* (*nat* (*a mod p*) $\hat{}$ *p*)
by *simp*
also have [... = *int* (*nat* (*a mod p*))] (*mod p*)
by (*subst cong-int-iff*, *rule little-Fermat-nat*) (*use assms not-dvd* **in** *auto*)
also have *int* (*nat* (*a mod p*)) = *a mod p*
using ‹$p > 1$› **by** *simp*
also have [*a mod p* = *a*] (*mod p*)
by (*auto simp*: *cong-def*)
finally show *?thesis* **.**
qed

lemma *prime-dvd-choose*:
assumes $0 < k$ $k < p$ *prime p*
shows *p dvd* (*p choose k*)
proof −
have $k \leq p$ **using** ‹$k < p$› **by** *auto*

have *p dvd fact p* **using** *assms* **by** (*simp add*: *prime-dvd-fact-iff*)

moreover have ¬ *p dvd fact k* ∗ *fact* (*p* − *k*)
unfolding *prime-dvd-mult-iff*[*OF assms*(*3*)] *prime-dvd-fact-iff*[*OF assms*(*3*)]
using *assms* **by** *simp*

ultimately show *?thesis*
unfolding *binomial-fact-lemma*[*OF* ‹$k \leq p$›, *symmetric*]
using *assms prime-dvd-multD* **by** *blast*
qed

lemma *prime-natD*:
assumes *prime* (*p* :: *nat*) *a dvd p*

shows $a = 1 \vee a = p$
using *assms* **by** (*auto simp*: *prime-nat-iff*)

lemma *not-prime-imp-ex-prod-nat*:
assumes $m > 1$ $\neg$ *prime* (m::*nat*)
shows $\exists n\ k.\ m = n * k \wedge 1 < n \wedge n < m \wedge 1 < k \wedge k < m$
proof −
from *assms* **have** ¬*Factorial-Ring.irreducible m*
by (*simp flip*: *prime-elem-iff-irreducible*)
with *assms* **obtain** $n\ k$ **where** *nk*: $m = n * k\ n \neq 1\ k \neq 1$
by (*auto simp*: *Factorial-Ring.irreducible-def*)
moreover from *this assms* **have** $n > 0\ k > 0$
by *auto*
with *nk* **have** $n > 1\ k > 1$ **by** *auto*
moreover {
from *assms nk* **have** n *dvd* $m\ k$ *dvd* m **by** *auto*
with *assms* **have** $n \leq m\ k \leq m$
by (*auto intro*!: *dvd-imp-le*)
moreover from *nk* ‹$n > 1$› ‹$k > 1$› **have** $n \neq m\ k \neq m$
by *auto*
ultimately have $n < m\ k < m$ **by** *auto*
}
ultimately show *?thesis* **by** *blast*
qed

1.2 Auxiliary algebraic material

lemma (**in** *group*) *ord-eqI-prime-factors*:
assumes $\bigwedge p.\ p \in$ *prime-factors* $n \Longrightarrow x\ [\hat{}]\ (n\ div\ p) \neq \mathbf{1}$ **and** $x\ [\hat{}]\ n = \mathbf{1}$
assumes $x \in$ *carrier* $G\ n > 0$
shows *group.ord* $G\ x = n$
proof −
have *group.ord* $G\ x$ *dvd* n
using *assms* **by** (*subst pow-eq-id* [*symmetric*]) *auto*
then obtain k **where** k: $n =$ *group.ord* $G\ x * k$
by *auto*
have $k = 1$
proof (*rule ccontr*)
assume $k \neq 1$
then obtain p **where** p: *prime* p p *dvd* k
using *prime-factor-nat* **by** *blast*
have $x\ [\hat{}]\ (group.ord\ G\ x * (k\ div\ p)) = \mathbf{1}$
by (*subst pow-eq-id*) (*use assms* **in** *auto*)
also have *group.ord* $G\ x * (k\ div\ p) = n\ div\ p$
using p **by** (*auto simp*: k)
finally have $x\ [\hat{}]\ (n\ div\ p) = \mathbf{1}$ **.**
moreover have $x\ [\hat{}]\ (n\ div\ p) \neq \mathbf{1}$
using $p\ k$ *assms* **by** (*intro assms*) (*auto simp*: *in-prime-factors-iff*)
ultimately show *False* **by** *contradiction*

```
  qed
  with k show ?thesis by simp
qed

lemma (in monoid) pow-nat-eq-1-imp-unit:
  fixes n :: nat
  assumes x [^] n = 1 and n > 0 and [simp]: x ∈ carrier G
  shows   x ∈ Units G
proof −
  from ‹n > 0› have x [^] (1 :: nat) ⊗ x [^] (n − 1) = x [^] n
    by (subst nat-pow-mult) auto
  with assms have x ⊗ x [^] (n − 1) = 1
    by simp
  moreover from ‹n > 0› have x [^] (n − 1) ⊗ x [^] (1 :: nat) = x [^] n
    by (subst nat-pow-mult) auto
  with assms have x [^] (n − 1) ⊗ x = 1
    by simp
  ultimately show ?thesis by (auto simp: Units-def)
qed

lemma (in cring) finsum-reindex-bij-betw:
  assumes bij-betw h S T g ∈ T → carrier R
  shows   finsum R (λx. g (h x)) S = finsum R g T
  using assms by (auto simp: bij-betw-def finsum-reindex)

lemma (in cring) finsum-reindex-bij-witness:
  assumes witness:
    ⋀a. a ∈ S ⟹ i (j a) = a
    ⋀a. a ∈ S ⟹ j a ∈ T
    ⋀b. b ∈ T ⟹ j (i b) = b
    ⋀b. b ∈ T ⟹ i b ∈ S
    ⋀b. b ∈ S ⟹ g b ∈ carrier R
  assumes eq:
    ⋀a. a ∈ S ⟹ h (j a) = g a
  shows finsum R g S = finsum R h T
proof −
  have bij: bij-betw j S T
    using bij-betw-byWitness[where A=S and f=j and f'=i and A'=T] witness
by auto
  hence T-eq: T = j ‘ S by (auto simp: bij-betw-def)
  from assms have h ∈ T → carrier R
    by (subst T-eq) auto
  moreover have finsum R g S = finsum R (λx. h (j x)) S
    using assms by (intro finsum-cong) (auto simp: eq)
  ultimately show ?thesis using assms(5)
    using finsum-reindex-bij-betw[OF bij, of h] by simp
qed

lemma (in cring) binomial:
```

fixes $n :: nat$
assumes [*simp*]: $x \in carrier\ R\ y \in carrier\ R$
shows $(x \oplus y)\ [\widehat{}]\ n = (\bigoplus i{\in}\{..n\}.\ add\text{-}pow\ R\ (n\ choose\ i)\ (x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (n - i)))$
proof (*induction n*)
case (*Suc n*)
have *binomial-Suc*: $Suc\ n\ choose\ i = (n\ choose\ (i - 1)) + (n\ choose\ i)$ **if** $i \in \{1..n\}$ **for** i
using *that* **by** (*cases i*) *auto*
have *Suc-diff*: $Suc\ n - i = Suc\ (n - i)$ **if** $i \leq n$ **for** i
using *that* **by** *linarith*
have $(x \oplus y)\ [\widehat{}]\ Suc\ n =$
$(\bigoplus i{\in}\{..n\}.\ add\text{-}pow\ R\ (n\ choose\ i)\ (x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (n - i))) \otimes x \oplus$
$(\bigoplus i{\in}\{..n\}.\ add\text{-}pow\ R\ (n\ choose\ i)\ (x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (n - i))) \otimes y$
by (*simp add*: *semiring-simprules Suc*)
also have $(\bigoplus i{\in}\{..n\}.\ add\text{-}pow\ R\ (n\ choose\ i)\ (x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (n - i))) \otimes x =$
$(\bigoplus i{\in}\{..n\}.\ add\text{-}pow\ R\ (n\ choose\ i)\ (x\ [\widehat{}]\ Suc\ i \otimes y\ [\widehat{}]\ (n - i)))$
by (*subst finsum-ldistr*)
(*auto simp*: *cring-simprules Suc add-pow-rdistr intro*!: *finsum-cong*)
also have $\ldots = (\bigoplus i{\in}\{1..Suc\ n\}.\ add\text{-}pow\ R\ (n\ choose\ (i - 1))\ (x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (Suc\ n - i)))$
by (*intro finsum-reindex-bij-witness*[*of* - $\lambda i.\ i - 1\ Suc$]) *auto*
also have $\{1..Suc\ n\} = insert\ (Suc\ n)\ \{1..n\}$ **by** *auto*
also have $(\bigoplus i{\in}\ldots.\ add\text{-}pow\ R\ (n\ choose\ (i - 1))\ (x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (Suc\ n - i))) =$
$x\ [\widehat{}]\ Suc\ n \oplus (\bigoplus i{\in}\{1..n\}.\ add\text{-}pow\ R\ (n\ choose\ (i - 1))\ (x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (Suc\ n - i)))$
(**is** - = - $\oplus$ *?S1*) **by** (*subst finsum-insert*) *auto*
also have $(\bigoplus i{\in}\{..n\}.\ add\text{-}pow\ R\ (n\ choose\ i)\ (x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (n - i))) \otimes y =$
$(\bigoplus i{\in}\{..n\}.\ add\text{-}pow\ R\ (n\ choose\ i)\ (x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (Suc\ n - i)))$
by (*subst finsum-ldistr*)
(*auto simp*: *cring-simprules Suc add-pow-rdistr Suc-diff intro*!: *finsum-cong*)
also have $\{..n\} = insert\ 0\ \{1..n\}$ **by** *auto*
also have $(\bigoplus i{\in}\ldots.\ add\text{-}pow\ R\ (n\ choose\ i)\ (x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (Suc\ n - i))) =$
$y\ [\widehat{}]\ Suc\ n \oplus (\bigoplus i{\in}\{1..n\}.\ add\text{-}pow\ R\ (n\ choose\ i)\ (x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (Suc\ n - i)))$
(**is** - = - $\oplus$ *?S2*) **by** (*subst finsum-insert*) *auto*
also have $(x\ [\widehat{}]\ Suc\ n \oplus ?S1) \oplus (y\ [\widehat{}]\ Suc\ n \oplus ?S2) =$
$x\ [\widehat{}]\ Suc\ n \oplus y\ [\widehat{}]\ Suc\ n \oplus (?S1 \oplus ?S2)$
by (*simp add*: *cring-simprules*)
also have $?S1 \oplus ?S2 = (\bigoplus i{\in}\{1..n\}.\ add\text{-}pow\ R\ (Suc\ n\ choose\ i)\ (x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (Suc\ n - i)))$
by (*subst finsum-addf* [*symmetric*], *simp*, *simp*, *rule finsum-cong'*)
(*auto intro*!: *finsum-cong simp*: *binomial-Suc add.nat-pow-mult*)
also have $x\ [\widehat{}]\ Suc\ n \oplus y\ [\widehat{}]\ Suc\ n \oplus \ldots =$
$(\bigoplus i{\in}\{0,\ Suc\ n\} \cup \{1..n\}.\ add\text{-}pow\ R\ (Suc\ n\ choose\ i)\ (x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (Suc\ n - i)))$
by (*subst finsum-Un-disjoint*) (*auto simp*: *cring-simprules*)
also have $\{0,\ Suc\ n\} \cup \{1..n\} = \{..Suc\ n\}$ **by** *auto*

finally show *?case* **.**
qed *auto*

lemma (**in** *cring*) *binomial-finite-char*:
fixes p :: *nat*
assumes [*simp*]: $x \in$ *carrier* R $y \in$ *carrier* R **and** *add-pow* R p $\mathbf{1} = \mathbf{0}$ *prime* p
shows $(x \oplus y)\ [\widehat{}]\ p = x\ [\widehat{}]\ p \oplus y\ [\widehat{}]\ p$
proof −
have $*$: *add-pow* R (p *choose* i) ($x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (p - i)$) $= \mathbf{0}$ **if** $i \in \{1..<p\}$ **for** i
proof −
have p *dvd* (p *choose* i)
by (*rule prime-dvd-choose*) (*use that assms* **in** *auto*)
then obtain k **where** [*simp*]: (p *choose* i) $= p * k$
by *auto*
have *add-pow* R (p *choose* i) ($x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (p - i)$) =
add-pow R (p *choose* i) $\mathbf{1} \otimes (x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (p - i))$
by (*simp add*: *add-pow-ldistr*)
also have *add-pow* R (p *choose* i) $\mathbf{1} = \mathbf{0}$
using *assms* **by** (*simp flip*: *add.nat-pow-pow*)
finally show *?thesis* **by** *simp*
qed

have $(x \oplus y)\ [\widehat{}]\ p = (\bigoplus i{\in}\{..p\}.$ *add-pow* R (p *choose* i) ($x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (p - i)$))
by (*rule binomial*) *auto*
also have $\ldots = (\bigoplus i{\in}\{0, p\}.$ *add-pow* R (p *choose* i) ($x\ [\widehat{}]\ i \otimes y\ [\widehat{}]\ (p - i)$))
using $*$ **by** (*intro add.finprod-mono-neutral-cong-right*) *auto*
also have $\ldots = x\ [\widehat{}]\ p \oplus y\ [\widehat{}]\ p$
using *assms prime-gt-0-nat*[*of* p] **by** (*simp add*: *cring-simprules*)
finally show *?thesis* **.**
qed

lemma (**in** *ring-hom-cring*) *hom-add-pow-nat*:
$x \in$ *carrier* $R \Longrightarrow h$ (*add-pow* R (n::*nat*) x) = *add-pow* S n (h x)
by (*induction* n) *auto*

end

2 The Lucas–Lehmer test

theory *Lucas-Lehmer*
imports
Lucas-Lehmer-Auxiliary
HOL−Algebra.Ring
Probabilistic-Prime-Tests.Jacobi-Symbol
Pell.Pell
begin

2.1 General properties of Mersenne numbers and Mersenne primes

We mostly follow the proofs given on Wikipedia [4, 3] in the following sections.

We first show some basic and theorems about Mersenne numbers and Mersenne primes in general, beginning with this: Mersenne primes are the only primes of the form $a^n - 1$ for $n > 1$.

```
lemma prime-power-minus-oneD:
  fixes a n :: nat
  assumes prime (a ^ n - 1)
  shows   n = 1 ∨ a = 2
proof -
  from assms have n > 0
    by (intro Nat.gr0I) auto
  have a ≠ 0 a ≠ 1
    by (rule notI, use ‹n > 0› assms in ‹simp add: zero-power›)+
  hence a > 1 by auto
  have [a - 1 + 1 = 0 + 1] (mod (a - 1))
    by (rule cong-add) (auto simp: cong-def)
  hence [a = 1] (mod (a - 1))
    using ‹a > 1› by simp
  hence [a ^ n - 1 = 1 ^ n - 1] (mod (a - 1))
    using ‹a > 1› by (intro cong-pow cong-diff-nat) auto
  hence (a - 1) dvd (a ^ n - 1)
    by (simp add: cong-0-iff)
  have a - 1 = 1 ∨ a - 1 = a ^ n - 1
    using ‹prime (a ^ n - 1)› and ‹(a - 1) dvd -› by (rule prime-natD)
  thus ?thesis
  proof
    assume a - 1 = 1
    hence a = 2 by simp
    thus ?thesis by simp
  next
    assume a - 1 = a ^ n - 1
    hence a ^ n = a ^ 1
      using ‹a > 1› by (simp add: Nat.eq-diff-iff)
    hence n = 1
      using ‹a > 1› by (subst (asm) power-inject-exp) auto
    thus ?thesis by simp
  qed
qed
```

Next, we show that if a prime q divides a Mersenne number $2^p - 1$ with an odd prime exponent p, then q must be of the form $q = 1 + 2kp$ for some $k > 0$.

```
lemma prime-dvd-mersenneD:
  fixes p q :: nat
```

```
  assumes prime p p ≠ 2 prime q q dvd (2 ^ p − 1)
  shows   [q = 1] (mod (2 * p))
proof −
  from assms have odd p
    using prime-gt-1-nat[of p] by (intro prime-odd-nat) auto
  have q ≠ 0 q ≠ 1 q ≠ 2
    using assms by (auto intro!: Nat.gr0I)
  hence q > 2 by simp
  with ‹prime q› have odd q
    by (simp add: prime-odd-nat)

  have ord q 2 = p
  proof −
    from assms have [2 ^ p − 1 + 1 = 0 + 1] (mod q)
      by (intro cong-add cong-refl) (auto simp: cong-0-iff)
    hence [2 ^ p = 1] (mod q) by simp
    hence ord q 2 dvd p
      by (subst (asm) ord-divides)
    hence ord q 2 = 1 ∨ ord q 2 = p
      using ‹prime p› and prime-natD by blast
    moreover have ord q 2 ≠ 1
      using ord-works[of 2 q] and ‹prime q› by (auto simp: cong-altdef-nat)
    ultimately show ord q 2 = p by blast
  qed

  have q-dvd-iff: q dvd (2 ^ x − 1) ⟷ p dvd x for x :: nat
  proof −
    have q dvd (2 ^ x − 1) ⟷ [2 ^ x = 1] (mod q)
      by (auto simp: cong-altdef-nat)
    also have ... ⟷ ord q 2 dvd x
      by (rule ord-divides)
    also note ‹ord q 2 = p›
    finally show ?thesis .
  qed

  from ‹q > 2› and assms have ¬q dvd 2
    using primes-dvd-imp-eq two-is-prime-nat by blast
  hence [2 ^ (q − 1) − 1 = 1 − 1] (mod q)
    using assms by (intro fermat-theorem cong-diff-nat) auto
  hence q dvd (2 ^ (q − 1) − 1)
    by (simp add: cong-0-iff)
  hence p dvd (q − 1)
    by (subst (asm) q-dvd-iff)
  hence [q = 1] (mod p)
    using ‹q > 2› by (auto simp: cong-altdef-nat prime-gt-1-nat)

  moreover have [q = 1] (mod 2)
    using ‹odd q› by (auto simp: cong-def odd-iff-mod-2-eq-one)
  ultimately show [q = 1] (mod (2 * p))
```

```
    using ‹odd p› by (intro coprime-cong-mult-nat) auto
qed

lemma prime-dvd-mersenneD':
  fixes p q :: nat
  assumes prime p p ≠ 2 prime q q dvd (2 ^ p − 1)
  shows ∃k>0. q = 1 + 2 * k * p
proof −
  have q ≠ 0 q ≠ 1 q ≠ 2
    using assms by (auto intro!: Nat.gr0I)
  hence q > 2 by simp

  have [q = 1] (mod (2 * p))
    by (rule prime-dvd-mersenneD) fact+
  hence (2 * p) dvd (q − 1)
    using ‹q > 2› by (auto simp: cong-altdef-nat)
  then obtain k where k: q − 1 = (2 * p) * k
    by blast
  hence q = 1 + 2 * k * p
    using ‹q > 2› by (simp add: algebra-simps)
  moreover have k > 0
    using ‹q > 2› and k by (intro Nat.gr0I) auto
  ultimately show ?thesis by blast
qed
```

A Mersenne number is any number of the form $2^p - 1$ for a natural number p. To make things a bit more pleasant, we additionally exclude $2^2 - 1$, i.e. we require $p > 2$. It can be shown that p is then always an odd prime.

```
locale mersenne-prime =
  fixes p M :: nat
  defines M ≡ 2 ^ p − 1
  assumes p-gt-2: p > 2 and prime: prime M
begin

lemma M-gt-6: M > 6
proof −
  from p-gt-2 have 2 ^ p ≥ (2 ^ 3 :: nat)
    by (intro power-increasing) auto
  thus ?thesis by (simp add: M-def)
qed

lemma M-odd: odd M
  using p-gt-2 by (auto simp: M-def)

theorem p-prime: prime p
proof (rule ccontr)
  assume ¬prime p
  then obtain a b where ab: p = a * b a > 1 b > 1
    using p-gt-2 not-prime-imp-ex-prod-nat[of p] by auto
```

```
  have geometric-sum-aux: (x − (1 :: int)) ∗ (∑ k<a. x ^ k) = x ^ a − 1 for x
    by (induction a) (auto simp: algebra-simps)
  have (2 ^ b − 1 :: int) ∗ (∑ k<a. (2 ^ b) ^ k) = (2 ^ b) ^ a − 1
    by (rule geometric-sum-aux)
  hence 2 ^ (a∗b) − 1 = (2 ^ b − 1 :: int) ∗ (∑ k<a. 2 ^ (k∗b))
    by (simp flip: power-mult add: algebra-simps)
  hence (2 ^ b − 1) dvd (2 ^ (a∗b) − 1 :: int)
    by simp
  hence int (2 ^ b − 1) dvd int (2 ^ (a ∗ b) − 1)
    by (subst of-nat-diff) (auto simp: of-nat-diff)
  hence (2 ^ b − 1) dvd (2 ^ (a ∗ b) − 1 :: nat)
    by (subst (asm) int-dvd-int-iff)
  with prime have 2 ^ b − 1 = (1 :: nat) ∨ 2 ^ b − 1 = (2 ^ p − 1 :: nat)
    unfolding ab M-def by (intro prime-natD) auto
  moreover have 2 ^ b > (2 ^ 1 :: nat)
    using ab by (intro power-strict-increasing) auto
  moreover have 2 ^ b < (2 ^ p :: nat)
    using ab by (intro power-strict-increasing) auto
  hence 2 ^ b − 1 < (2 ^ p − 1 :: nat)
    by (subst less-diff-iff) auto
  ultimately show False by auto
qed

lemma p-odd: odd p
  using p-prime p-gt-2 prime-odd-nat by auto
```

We now first show a few more properties of Mersenne primes regarding congruences and the Legendre symbol.

```
lemma M-cong-7-mod-12: [M = 7] (mod 12)
proof −
  have [M = 8 − 1] (mod 12)
   using p-gt-2 p-odd unfolding M-def by (intro cong-diff-nat two-power-odd-mod-12)
auto
  thus [M = 7] (mod 12) by simp
qed

lemma Legendre-3-M: Legendre 3 M = −1
  using prime M-cong-7-mod-12 by (subst Legendre-3-left) (auto simp: cong-def)

lemma M-cong-7-mod-8: [M = 7] (mod 8)
proof −
  have 2 ^ 3 dvd (2 ^ p :: int)
    using p-gt-2 by (intro le-imp-power-dvd) auto
  hence [2 ^ p − 1 = 0 − 1] (mod (8 :: int))
    by (intro cong-diff) (auto simp: cong-def)
  also have 2 ^ p − 1 = int M
    by (simp add: M-def of-nat-diff)
  finally have int M mod int 8 = 7
```

```
    by (simp add: cong-def)
  thus [M = 7] (mod 8)
    by (subst (asm) zmod-int [symmetric]) (auto simp: cong-def)
qed

lemma Legendre-2-M: Legendre 2 M = 1
  using prime M-gt-6 M-cong-7-mod-8
  by (subst supplement2-Legendre') (auto simp: cong-def nat-mod-as-int)

lemma M-not-dvd-24: ¬M dvd 24
proof
  assume M dvd 24
  hence M dvd 2 * 2 * 2 * 3
    by simp
  also have ?this ⟷ M dvd 2 ∨ M dvd 3
    using prime by (simp only: prime-dvd-mult-iff) auto
  finally show False using M-gt-6 by (auto dest: dvd-imp-le)
qed

end
```

2.2 The Lucas–Lehmer sequence

We now define the Lucas–Lehmer sequence $a_{n+1} = a_n^2 - 2$. The starting value we will always use is $a_0 = 4$.

```
primrec gen-lucas-lehmer-sequence :: int ⇒ nat ⇒ int where
  gen-lucas-lehmer-sequence a 0 = a
| gen-lucas-lehmer-sequence a (Suc n) = gen-lucas-lehmer-sequence a n ^ 2 - 2

lemma gen-lucas-lehmer-sequence-Suc':
  gen-lucas-lehmer-sequence a (Suc n) = gen-lucas-lehmer-sequence (a ^ 2 - 2) n
  by (induction n arbitrary: a) auto

lemmas gen-lucas-lehmer-code [code] =
  gen-lucas-lehmer-sequence.simps(1) gen-lucas-lehmer-sequence-Suc'
```

For $a_0 = 4$, the recurrence has the closed form $a_{4,n} = \omega^{2^n} + \bar{\omega}^{2^n}$ with $\omega = 2 + \sqrt{3}$ and $\bar{\omega} = 2 - \sqrt{3}$.

```
lemma gen-lucas-lehmer-sequence-4-closed-form1:
  real-of-int (gen-lucas-lehmer-sequence 4 n) = (2 + sqrt 3) ^ (2 ^ n) + (2 - sqrt
3) ^ (2 ^ n)
  by (induction n)
    (auto simp: algebra-simps power2-eq-square power-mult simp flip: power-mult-distrib)

lemma gen-lucas-lehmer-sequence-4-closed-form2:
  gen-lucas-lehmer-sequence 4 n = round ((2 + sqrt 3) ^ (2 ^ n))
proof (rule sym, rule round-unique')
  have 5 / 3 < sqrt (3 :: real)
```

by (*rule real-less-rsqrt*) (*auto simp*: *power2-eq-square*)
hence (*2 − sqrt 3*) ^ (*2 ^ n*) < (*1 / 3*) ^ (*2 ^ n*)
by (*intro power-strict-mono*) (*auto simp*: *real-le-lsqrt*)
also have ... ≤ (*1 / 3*) ^ *1*
by (*intro power-decreasing*) *auto*
finally have (*2 − sqrt 3*) ^ (*2 ^ n*) < *1 / 2* **by** *simp*
moreover have (*2 − sqrt 3*) ^ (*2 ^ n*) ≥ *0*
by (*intro zero-le-power*) (*auto simp*: *real-le-lsqrt*)
ultimately show |(*2 + sqrt 3*) ^ *2 ^ n − real-of-int* (*gen-lucas-lehmer-sequence 4 n*)| < *1 / 2*
unfolding *gen-lucas-lehmer-sequence-4-closed-form1* **by** *linarith*
qed

lemma *gen-lucas-lehmer-sequence-4-closed-form3*:
gen-lucas-lehmer-sequence 4 n = ⌈(*2 + sqrt 3*) ^ (*2 ^ n*)⌉
proof (*rule sym*, *rule ceiling-unique*)
show *real-of-int* (*gen-lucas-lehmer-sequence 4 n*) ≥ (*2 + sqrt 3*) ^ *2 ^ n*
unfolding *gen-lucas-lehmer-sequence-4-closed-form1* **by** (*auto intro*!: *zero-le-power real-le-lsqrt*)
next
have *5 / 3* < *sqrt* (*3* :: *real*)
by (*rule real-less-rsqrt*) (*auto simp*: *power2-eq-square*)
hence (*2 − sqrt 3*) ^ (*2 ^ n*) < (*1 / 3*) ^ (*2 ^ n*)
by (*intro power-strict-mono*) (*auto simp*: *real-le-lsqrt*)
also have ... ≤ (*1 / 3*) ^ *1*
by (*intro power-decreasing*) *auto*
finally have (*2 − sqrt 3*) ^ (*2 ^ n*) < *1 / 2* **by** *simp*
moreover have (*2 − sqrt 3*) ^ (*2 ^ n*) ≥ *0*
by (*intro zero-le-power*) (*auto simp*: *real-le-lsqrt*)
ultimately show *real-of-int* (*gen-lucas-lehmer-sequence 4 n*) − *1* < (*2 + sqrt 3*) ^ *2 ^ n*
unfolding *gen-lucas-lehmer-sequence-4-closed-form1* **by** *linarith*
qed

2.3 The ring $\mathbb{Z}[\sqrt{3}]$

To relate this sequence to Mersenne primes, we now first need to define the ring $\mathbb{Z}[\sqrt{3}]$, which is a subring of $\mathbb{R}$. This ring can be seen as the lattice on $\mathbb{R}$ that is freely generated by 1 and $\sqrt{3}$.

It is, however, more convenient to explicitly describe it as a ring structure over the set $\mathbb{Z}\times\mathbb{Z}$ with a corresponding injective homomorphism $\mathbb{Z}\times\mathbb{Z} \to \mathbb{R}$.

definition *lucas-lehmer-add′* :: *int* × *int* ⇒ *int* × *int* ⇒ *int* × *int* **where**
lucas-lehmer-add′ = (λ(*a*,*b*) (*c*,*d*). (*a + c*, *b + d*))

definition *lucas-lehmer-mult′* :: *int* × *int* ⇒ *int* × *int* ⇒ *int* × *int* **where**
lucas-lehmer-mult′ = (λ(*a*,*b*) (*c*,*d*). (*a * c + 3 * b * d*, *a * d + b * c*))

definition *lucas-lehmer-ring* :: (*int* × *int*) *ring* **where**

lucas-lehmer-ring =
(⦇*carrier* = *UNIV*,
monoid.mult = *lucas-lehmer-mult'*,
one = (*1*, *0*),
ring.zero = (*0*, *0*),
add = *lucas-lehmer-add'*⦈)

lemma *carrier-lucas-lehmer-ring* [*simp*]: *carrier lucas-lehmer-ring* = *UNIV*
by (*simp add*: *lucas-lehmer-ring-def*)

lemma *cring-lucas-lehmer-ring* [*intro*]: *cring* (*lucas-lehmer-ring*)
proof
have ∃ *aa ba*. *lucas-lehmer-add'* (*aa*, *ba*) (*a*, *b*) = (*0*, *0*) ∧
lucas-lehmer-add' (*a*, *b*) (*aa*, *ba*) = (*0*, *0*) **for** *a b*
by (*rule exI*[*of* - −*a*], *rule exI*[*of* - −*b*]) (*auto simp*: *lucas-lehmer-add'-def*)
thus *carrier* (*add-monoid lucas-lehmer-ring*) ⊆ *Units* (*add-monoid lucas-lehmer-ring*)
by (*auto simp*: *Units-def lucas-lehmer-ring-def*)
qed (*auto simp*: *lucas-lehmer-ring-def lucas-lehmer-add'-def lucas-lehmer-mult'-def algebra-simps*)

2.4 The ring $(\mathbb{Z}/m\mathbb{Z})[\sqrt{3}]$

We shall also need the ring $(\mathbb{Z}/m\mathbb{Z})[\sqrt{3}]$, which is obtained from $\mathbb{Z}[\sqrt{3}]$ by reducing each component separately modulo m. This essentially identifies any two points that are a multiple of m apart and then all those that are a multiple of $m\sqrt{3}$ apart.

definition *lucas-lehmer-mult* :: *nat* ⇒ *nat* × *nat* ⇒ *nat* × *nat* ⇒ *nat* × *nat*
where
lucas-lehmer-mult m = (λ(*a*,*b*) (*c*,*d*). ((*a* ∗ *c* + *3* ∗ *b* ∗ *d*) *mod m*, (*a* ∗ *d* + *b* ∗ *c*) *mod m*))

definition *lucas-lehmer-add* :: *nat* ⇒ *nat* × *nat* ⇒ *nat* × *nat* ⇒ *nat* × *nat* **where**
lucas-lehmer-add m = (λ(*a*,*b*) (*c*,*d*). ((*a* + *c*) *mod m*, (*b* + *d*) *mod m*))

definition *lucas-lehmer-ring-mod* :: *nat* ⇒ (*nat* × *nat*) *ring* **where**
lucas-lehmer-ring-mod m =
(⦇*carrier* = {..<*m*} × {..<*m*},
monoid.mult = *lucas-lehmer-mult m*,
one = (*1*, *0*),
ring.zero = (*0*, *0*),
add = *lucas-lehmer-add m*⦈)

lemma *lucas-lehmer-add-in-carrier*: *m* > *0* ⟹ *lucas-lehmer-add m x y* ∈ {..<*m*} × {..<*m*}
by (*auto simp*: *lucas-lehmer-add-def split*: *prod.splits*)

lemma *lucas-lehmer-mult-in-carrier*: *m* > *0* ⟹ *lucas-lehmer-mult m x y* ∈ {..<*m*} × {..<*m*}

by (*auto simp*: *lucas-lehmer-mult-def split*: *prod.splits*)

lemma *lucas-lehmer-add-cong*:
[*fst* (*lucas-lehmer-add m x y*) = *fst x* + *fst y*] (*mod m*)
[*snd* (*lucas-lehmer-add m x y*) = *snd x* + *snd y*] (*mod m*)
by (*simp-all add*: *lucas-lehmer-add-def cong-def case-prod-unfold*)

lemma *lucas-lehmer-mult-cong*:
[*fst* (*lucas-lehmer-mult m x y*) = *fst x* ∗ *fst y* + *3* ∗ *snd x* ∗ *snd y*] (*mod m*)
[*snd* (*lucas-lehmer-mult m x y*) = *fst x* ∗ *snd y* + *snd x* ∗ *fst y*] (*mod m*)
by (*simp-all add*: *lucas-lehmer-mult-def cong-def case-prod-unfold*)

lemma *lucas-lehmer-add-neutral* [*simp*]:
assumes *fst x* < *m snd x* < *m*
shows *lucas-lehmer-add m* (*0*, *0*) *x* = *x*
and *lucas-lehmer-add m x* (*0*, *0*) = *x*
using *assms* **by** (*auto simp*: *lucas-lehmer-add-def case-prod-unfold*)

lemma *lucas-lehmer-mult-neutral* [*simp*]:
assumes *fst x* < *m snd x* < *m*
shows *lucas-lehmer-mult m* (*Suc 0*, *0*) *x* = *x*
and *lucas-lehmer-mult m x* (*Suc 0*, *0*) = *x*
using *assms* **by** (*auto simp*: *lucas-lehmer-mult-def case-prod-unfold*)

lemma *lucas-lehmer-add-commute*: *lucas-lehmer-add m x y* = *lucas-lehmer-add m y x*
by (*simp add*: *lucas-lehmer-add-def algebra-simps case-prod-unfold*)

lemma *lucas-lehmer-mult-commute*: *lucas-lehmer-mult m x y* = *lucas-lehmer-mult m y x*
by (*simp add*: *lucas-lehmer-mult-def algebra-simps case-prod-unfold*)

lemma *lucas-lehmer-add-assoc*:
assumes *m*: *m* > *0*
shows *lucas-lehmer-add m x* (*lucas-lehmer-add m y z*) =
lucas-lehmer-add m (*lucas-lehmer-add m x y*) *z*
proof (*rule prod-eqI*)
let *?add* = *lucas-lehmer-add m*
have [*fst* (*?add x* (*?add y z*)) = *fst x* + (*fst y* + *fst z*)] (*mod m*)
by (*rule lucas-lehmer-add-cong*[*THEN cong-trans*] *cong-add cong-mult cong-refl*)+
also have *fst x* + (*fst y* + *fst z*) = (*fst x* + *fst y*) + *fst z*
by (*simp add*: *add-ac*)
also have [. . . = *fst* (*?add* (*?add x y*) *z*)] (*mod m*)
by (*rule cong-sym*, (*rule lucas-lehmer-add-cong*[*THEN cong-trans*] *cong-add cong-mult cong-refl*)+)
finally show *fst* (*?add x* (*?add y z*)) = *fst* (*?add* (*?add x y*) *z*)
by (*rule cong-less-modulus-unique-nat*)
(*use m* **in** ‹*auto simp*: *lucas-lehmer-add-def case-prod-unfold*›)

have [*snd* (*?add x* (*?add y z*)) = *snd x* + (*snd y* + *snd z*)] (*mod m*)
by (*rule lucas-lehmer-add-cong*[*THEN cong-trans*] *cong-add cong-mult cong-refl*)+
also have *snd x* + (*snd y* + *snd z*) = (*snd x* + *snd y*) + *snd z*
by (*simp add*: *add-ac*)
also have [. . . = *snd* (*?add* (*?add x y*) *z*)] (*mod m*)
by (*rule cong-sym*, (*rule lucas-lehmer-add-cong*[*THEN cong-trans*] *cong-add cong-mult cong-refl*)+)
finally show *snd* (*?add x* (*?add y z*)) = *snd* (*?add* (*?add x y*) *z*)
by (*rule cong-less-modulus-unique-nat*)
(*use m* **in** ‹*auto simp*: *lucas-lehmer-add-def case-prod-unfold*›)
qed

lemma *lucas-lehmer-mult-assoc*:
assumes *m*: *m* > *0*
shows *lucas-lehmer-mult m x* (*lucas-lehmer-mult m y z*) =
lucas-lehmer-mult m (*lucas-lehmer-mult m x y*) *z*
proof (*rule prod-eqI*)
let *?mul* = *lucas-lehmer-mult m*
have [*fst* (*?mul x* (*?mul y z*)) = *fst x* ∗ (*fst y* ∗ *fst z* + *3* ∗ *snd y* ∗ *snd z*) +
3 ∗ *snd x* ∗ (*fst y* ∗ *snd z* + *snd y* ∗ *fst z*)] (*mod m*)
by (*rule lucas-lehmer-mult-cong*[*THEN cong-trans*] *cong-add cong-mult cong-refl*)+
also have *fst x* ∗ (*fst y* ∗ *fst z* + *3* ∗ *snd y* ∗ *snd z*) +
3 ∗ *snd x* ∗ (*fst y* ∗ *snd z* + *snd y* ∗ *fst z*) =
(*fst x* ∗ *fst y* + *3* ∗ *snd x* ∗ *snd y*) ∗ *fst z* +
3 ∗ (*fst x* ∗ *snd y* + *snd x* ∗ *fst y*) ∗ *snd z*
by (*simp add*: *algebra-simps*)
also have [. . . = *fst* (*?mul* (*?mul x y*) *z*)] (*mod m*)
by (*rule cong-sym*, (*rule lucas-lehmer-mult-cong*[*THEN cong-trans*] *cong-add cong-mult cong-refl*)+)
finally show *fst* (*?mul x* (*?mul y z*)) = *fst* (*?mul* (*?mul x y*) *z*)
by (*rule cong-less-modulus-unique-nat*)
(*use m* **in** ‹*auto simp*: *lucas-lehmer-mult-def case-prod-unfold*›)

have [*snd* (*?mul x* (*?mul y z*)) = *fst x* ∗ (*fst y* ∗ *snd z* + *snd y* ∗ *fst z*) +
snd x ∗ (*fst y* ∗ *fst z* + *3* ∗ *snd y* ∗ *snd z*)] (*mod m*)
by (*rule lucas-lehmer-mult-cong*[*THEN cong-trans*] *cong-add cong-mult cong-refl*)+
also have *fst x* ∗ (*fst y* ∗ *snd z* + *snd y* ∗ *fst z*) + *snd x* ∗ (*fst y* ∗ *fst z* + *3* ∗ *snd y* ∗ *snd z*) =
(*fst x* ∗ *fst y* + *3* ∗ *snd x* ∗ *snd y*) ∗ *snd z* + (*fst x* ∗ *snd y* + *snd x* ∗ *fst y*) ∗ *fst z*
by (*simp add*: *algebra-simps*)
also have [. . . = *snd* (*?mul* (*?mul x y*) *z*)] (*mod m*)
by (*rule cong-sym*, (*rule lucas-lehmer-mult-cong*[*THEN cong-trans*] *cong-add cong-mult cong-refl*)+)
finally show *snd* (*?mul x* (*?mul y z*)) = *snd* (*?mul* (*?mul x y*) *z*)
by (*rule cong-less-modulus-unique-nat*)
(*use m* **in** ‹*auto simp*: *lucas-lehmer-mult-def case-prod-unfold*›)
qed

lemma *lucas-lehmer-distrib-right*:
assumes *m*: *m* > *1*
shows *lucas-lehmer-mult m* (*lucas-lehmer-add m x y*) *z* =
lucas-lehmer-add m (*lucas-lehmer-mult m x z*) (*lucas-lehmer-mult m y z*)
proof (*rule prod-eqI*)
let *?mul* = *lucas-lehmer-mult m* **and** *?add* = *lucas-lehmer-add m*
have [*fst* (*?mul* (*?add x y*) *z*) = (*fst x* + *fst y*) ∗ *fst z* + *3* ∗ (*snd x* + *snd y*) ∗ *snd z*] (*mod m*)
by (*rule lucas-lehmer-mult-cong*[*THEN cong-trans*] *lucas-lehmer-add-cong*[*THEN cong-trans*]
cong-add cong-mult cong-refl)+
also have (*fst x* + *fst y*) ∗ *fst z* + *3* ∗ (*snd x* + *snd y*) ∗ *snd z* =
(*fst x* ∗ *fst z* + *3* ∗ *snd x* ∗ *snd z*) + (*fst y* ∗ *fst z* + *3* ∗ *snd y* ∗ *snd z*)
by (*simp add*: *algebra-simps*)
also have [. . . = *fst* (*?add* (*?mul x z*) (*?mul y z*))] (*mod m*)
by (*rule cong-sym*, (*rule lucas-lehmer-mult-cong*[*THEN cong-trans*]
lucas-lehmer-add-cong[*THEN cong-trans*] *cong-add cong-mult cong-refl*)+)
finally show *fst* (*?mul* (*?add x y*) *z*) = *fst* (*?add* (*?mul x z*) (*?mul y z*))
by (*rule cong-less-modulus-unique-nat*)
(*use m* **in** ‹*auto simp*: *lucas-lehmer-add-def lucas-lehmer-mult-def case-prod-unfold*›)

have [*snd* (*?mul* (*?add x y*) *z*) = (*fst x* + *fst y*) ∗ *snd z* + (*snd x* + *snd y*) ∗ *fst z*] (*mod m*)
by (*rule lucas-lehmer-mult-cong*[*THEN cong-trans*] *lucas-lehmer-add-cong*[*THEN cong-trans*]
cong-add cong-mult cong-refl)+
also have (*fst x* + *fst y*) ∗ *snd z* + (*snd x* + *snd y*) ∗ *fst z* =
(*fst x* ∗ *snd z* + *snd x* ∗ *fst z*) + (*fst y* ∗ *snd z* + *snd y* ∗ *fst z*)
by (*simp add*: *algebra-simps*)
also have [. . . = *snd* (*?add* (*?mul x z*) (*?mul y z*))] (*mod m*)
by (*rule cong-sym*, (*rule lucas-lehmer-mult-cong*[*THEN cong-trans*]
lucas-lehmer-add-cong[*THEN cong-trans*] *cong-add cong-mult cong-refl*)+)
finally show *snd* (*?mul* (*?add x y*) *z*) = *snd* (*?add* (*?mul x z*) (*?mul y z*))
by (*rule cong-less-modulus-unique-nat*)
(*use m* **in** ‹*auto simp*: *lucas-lehmer-add-def lucas-lehmer-mult-def case-prod-unfold*›)
qed

lemma *lucas-lehmer-distrib-left*:
assumes *m* > *1*
shows *lucas-lehmer-mult m z* (*lucas-lehmer-add m x y*) =
lucas-lehmer-add m (*lucas-lehmer-mult m z x*) (*lucas-lehmer-mult m z y*)
using *lucas-lehmer-distrib-right*[*of m x y z*] *assms*
by (*simp add*: *lucas-lehmer-mult-commute*)

lemma *cring-lucas-lehmer-ring-mod* [*intro*]:
assumes *m* > *1*
shows *cring* (*lucas-lehmer-ring-mod m*)
proof *unfold-locales*
let *?neg* = λ*x*. *if x* = *0 then 0 else m* − *x*

```
  have ∃x∈carrier (lucas-lehmer-ring-mod m).
         x ⊕lucas-lehmer-ring-mod m (a, b) = 0lucas-lehmer-ring-mod m ∧
         (a, b) ⊕lucas-lehmer-ring-mod m x = 0lucas-lehmer-ring-mod m
    if (a, b) ∈ carrier (lucas-lehmer-ring-mod m) for a b
    using that assms
    by (intro bexI[of - (?neg a, ?neg b)])
       (auto simp: lucas-lehmer-ring-mod-def lucas-lehmer-add-def)
  thus carrier (add-monoid (lucas-lehmer-ring-mod m)) ⊆ Units (add-monoid
(lucas-lehmer-ring-mod m))
    by (auto simp: Units-def)
qed (insert assms,
     auto simp: lucas-lehmer-ring-mod-def algebra-simps lucas-lehmer-mult-assoc
           lucas-lehmer-add-assoc lucas-lehmer-distrib-right lucas-lehmer-distrib-left
         intro: lucas-lehmer-mult-in-carrier lucas-lehmer-add-in-carrier
              lucas-lehmer-add-commute lucas-lehmer-mult-commute)
```

Since 0 is clearly not a unit in the ring and its carrier has size m^2, the number of units is strictly less than m^2.

```
lemma card-lucas-lehmer-Units:
  assumes m > 1
  shows   card (Units (lucas-lehmer-ring-mod m)) < m ^ 2
proof −
  interpret cring lucas-lehmer-ring-mod m
    using assms by auto
  have m ^ 2 > 0
    using assms by auto
  from assms have card (Units (lucas-lehmer-ring-mod m)) ≤ card ({..<m} ×
{..<m} − {(0, 0)})
    by (intro card-mono) (auto simp: Units-def lucas-lehmer-ring-mod-def lucas-lehmer-mult-def)
  also have ... = m ^ 2 − 1
    using assms by (subst card-Diff-subset) (auto simp: power2-eq-square)
  finally show ?thesis using ‹m ^ 2 > 0› by linarith
qed
```

Consider now the case of a prime modulus m: Since $\mathbb{Z}/m\mathbb{Z} = \mathrm{GF}(m)$ is a field, any element of $\mathbb{Z}/m\mathbb{Z}$ is a unit in $(\mathbb{Z}/m\mathbb{Z})[\sqrt{3}]$.

```
lemma int-in-Units-lucas-lehmer-ring-mod:
  assumes prime p
  assumes x > 0 x < p
  shows   (x, 0) ∈ Units (lucas-lehmer-ring-mod p)
proof −
  define R where R = lucas-lehmer-ring-mod p
  have [x * (x ^ (p − 2) mod p) = x * x ^ (p − 2)] (mod p)
    by (intro cong-mult) (auto simp: cong-def)
  also have x * x ^ (p − 2) = x ^ (Suc (p − 2))
    by (simp add: mult-ac)
  also have Suc (p − 2) = p − 1
    using prime-gt-1-nat[of p] assms by simp
  also have [x ^ (p − 1) = 1] (mod p)
```

using *assms* **by** (*intro fermat-theorem*) (*auto dest*: *dvd-imp-le*)
finally have $(x,\ 0) \otimes_R (x\ \widehat{}\ (p - 2)\ mod\ p,\ 0) = \mathbf{1}_R$
$(x\ \widehat{}\ (p - 2)\ mod\ p,\ 0) \otimes_R (x,\ 0) = \mathbf{1}_R$
$(x\ \widehat{}\ (p - 2)\ mod\ p,\ 0) \in carrier\ R$
using *prime-gt-1-nat*[*of p*] *assms*
by (*auto simp*: *lucas-lehmer-mult-def cong-def lucas-lehmer-ring-mod-def mult-ac R-def*)
moreover from *assms* **have** $(x,\ 0) \in carrier\ R$
by (*auto simp*: *R-def lucas-lehmer-ring-mod-def*)
ultimately show *?thesis* **using** *assms*
by (*auto simp*: *Units-def R-def*)
qed

2.5 $\mathbb{Z}[\sqrt{3}]$ as a subring of $\mathbb{R}$

We now define the homomorphism from $\mathbb{Z}[\sqrt{3}]$ into the reals:

definition *lucas-lehmer-to-real* :: *int* × *int* ⇒ *real* **where**
lucas-lehmer-to-real = (λ(*a*,*b*). *real-of-int a* + *real-of-int b* ∗ *sqrt 3*)

context
begin

interpretation *cring lucas-lehmer-ring* **..**

lemma *minus-lucas-lehmer-ring*: $\ominus_{lucas\text{-}lehmer\text{-}ring}\ x = (case\ x\ of\ (a,\ b) \Rightarrow (-a, -b))$
by (*rule sym*, *rule sum-zero-eq-neg*)
(*auto simp*: *case-prod-unfold lucas-lehmer-ring-def lucas-lehmer-add'-def*)

lemma *lucas-lehmer-to-real-simps1*:
lucas-lehmer-to-real $(a,\ b)$ = *of-int a* + *of-int b* ∗ *sqrt 3*
lucas-lehmer-to-real $(x \oplus_{lucas\text{-}lehmer\text{-}ring} y)$ =
lucas-lehmer-to-real x + *lucas-lehmer-to-real y*
lucas-lehmer-to-real $(x \otimes_{lucas\text{-}lehmer\text{-}ring} y)$ =
lucas-lehmer-to-real x ∗ *lucas-lehmer-to-real y*
lucas-lehmer-to-real $(\ominus_{lucas\text{-}lehmer\text{-}ring}\ x) = -lucas\text{-}lehmer\text{-}to\text{-}real\ x$
lucas-lehmer-to-real $(\mathbf{0}_{lucas\text{-}lehmer\text{-}ring}) = 0$
lucas-lehmer-to-real $(\mathbf{1}_{lucas\text{-}lehmer\text{-}ring}) = 1$
using *minus-lucas-lehmer-ring*
by (*simp-all add*: *lucas-lehmer-to-real-def lucas-lehmer-add'-def lucas-lehmer-mult'-def case-prod-unfold algebra-simps lucas-lehmer-ring-def*)

lemma *lucas-lehmer-to-add-pow-nat*:
lucas-lehmer-to-real $([n] \cdot_{lucas\text{-}lehmer\text{-}ring} x)$ = *of-nat n* ∗ *lucas-lehmer-to-real x*
by (*induction n*) (*auto simp*: *lucas-lehmer-to-real-simps1 algebra-simps*)

lemma *lucas-lehmer-to-add-pow-int*:
lucas-lehmer-to-real $([n] \cdot_{lucas\text{-}lehmer\text{-}ring} x)$ = *of-int n* ∗ *lucas-lehmer-to-real x*

proof (*cases* $n \geq 0$)
case *True*
hence *lucas-lehmer-to-real* ($[n]$ $\cdot_{lucas\text{-}lehmer\text{-}ring}$ x) =
lucas-lehmer-to-real ($[int\ (nat\ n)]$ $\cdot_{lucas\text{-}lehmer\text{-}ring}$ x)
by *simp*
also have ... = *lucas-lehmer-to-real* ($[nat\ n]$ $\cdot_{lucas\text{-}lehmer\text{-}ring}$ x)
by (*simp add*: *add-pow-int-ge*)
also have ... = *of-int* n $*$ *lucas-lehmer-to-real* x **using** *True*
by (*simp add*: *lucas-lehmer-to-add-pow-nat algebra-simps*)
finally show *?thesis* **.**
next
case *False*
hence *lucas-lehmer-to-real* ($[n]$ $\cdot_{lucas\text{-}lehmer\text{-}ring}$ x) =
lucas-lehmer-to-real (*add-pow lucas-lehmer-ring* ($-int\ (nat\ (-n))$) x)
by *simp*
also have *add-pow lucas-lehmer-ring* ($-int\ (nat\ (-n))$) x =
$\ominus_{lucas\text{-}lehmer\text{-}ring}$ (*add-pow lucas-lehmer-ring* ($nat\ (-n)$) x)
using *False* **by** (*subst add.int-pow-neg-int*) (*auto simp*: *lucas-lehmer-ring-def*)
also have *lucas-lehmer-to-real* ... = *of-int* n $*$ *lucas-lehmer-to-real* x **using** *False*
by (*simp add*: *lucas-lehmer-to-add-pow-nat lucas-lehmer-to-real-simps1 algebra-simps*)
finally show *?thesis* **.**
qed

lemma *lucas-lehmer-to-real-power*:
lucas-lehmer-to-real (x $[\widehat{}]_{lucas\text{-}lehmer\text{-}ring}$ (n :: *nat*)) = *lucas-lehmer-to-real* x $\widehat{}$ n
by (*induction n*) (*auto simp*: *lucas-lehmer-to-real-simps1*)

lemmas *lucas-lehmer-to-real-simps* =
lucas-lehmer-to-real-simps1 lucas-lehmer-to-real-power
lucas-lehmer-to-add-pow-nat lucas-lehmer-to-add-pow-int

end

lemma *lucas-lehmer-to-real-inj*: *inj lucas-lehmer-to-real*
proof (*rule injI*, *clarify*)
fix a b c d :: *int*
assume *eq*: *lucas-lehmer-to-real* (a, b) = *lucas-lehmer-to-real* (c, d)
have $b = d$
proof (*rule ccontr*)
assume $b \neq d$
hence *sqrt 3* = $(c - a)$ / $(b - d)$
using *eq* **by** (*simp add*: *lucas-lehmer-to-real-def field-simps*)
also have ... $\in \mathbb{Q}$ **by** *auto*
finally have *sqrt 3* $\in \mathbb{Q}$ **.**
moreover have *sqrt 3* $\notin \mathbb{Q}$
using *is-nth-power-prime-power-nat-iff*[*of 3 2 1*] *irrat-sqrt-nonsquare*[*of 3*] **by** *auto*

ultimately show *False* **by** *contradiction*
qed
moreover from *this* **and** *eq* **have** $a = c$
by (*auto simp*: *lucas-lehmer-to-real-def*)
ultimately show $a = c \wedge b = d$ **by** *blast*
qed

2.6 The canonical homomorphism $\mathbb{Z}[\sqrt{3}] \to (\mathbb{Z}/m\mathbb{Z})[\sqrt{3}]$

Next, we show that reduction modulo m is indeed a homomorphism.

definition *lucas-lehmer-hom* :: *nat* ⇒ (*int* × *int*) ⇒ (*nat* × *nat*) **where**
lucas-lehmer-hom m = (λ(*x*,*y*). (*nat* (*x mod m*), *nat* (*y mod m*)))

lemma *lucas-lehmer-hom-cong*:
[*fst x* = *fst y*] (*mod int m*) ⟹ [*snd x* = *snd y*] (*mod int m*) ⟹
lucas-lehmer-hom m x = *lucas-lehmer-hom m y*
by (*auto simp*: *lucas-lehmer-hom-def cong-def case-prod-unfold*)

lemma *lucas-lehmer-hom-cong′*:
[*a* = *b*] (*mod int m*) ⟹ [*c* = *d*] (*mod int m*) ⟹
lucas-lehmer-hom m (*a*, *c*) = *lucas-lehmer-hom m* (*b*, *d*)
by (*auto simp*: *lucas-lehmer-hom-def cong-def*)

context
fixes *m* :: *nat*
assumes *m*: *m* > *1*
begin

lemma *lucas-lehmer-hom-in-carrier*: *lucas-lehmer-hom m x* ∈ {..<*m*} × {..<*m*}
using *m nat-less-iff* **by** (*auto simp*: *lucas-lehmer-hom-def case-prod-unfold*)

lemma *lucas-lehmer-hom-add*:
lucas-lehmer-hom m (*lucas-lehmer-add′ x y*) =
lucas-lehmer-add m (*lucas-lehmer-hom m x*) (*lucas-lehmer-hom m y*)
proof (*rule prod-eqI*)
let *?add1* = *lucas-lehmer-add′* **and** *?add2* = *lucas-lehmer-add m*
let *?φ* = *lucas-lehmer-hom m*
have *fst* (*?φ* (*?add1 x y*)) = *nat* ((*fst x* + *fst y*) *mod int m*)
by (*simp add*: *lucas-lehmer-hom-def lucas-lehmer-add′-def case-prod-unfold*)
also have (*fst x* + *fst y*) *mod int m* = ((*fst x mod m*) + (*fst y mod m*)) *mod int m*
by (*simp add*: *mod-add-eq*)
also have *nat* ... = (*nat* (*fst x mod int m*) + *nat* (*fst y mod int m*)) *mod m*
using *m nat-add-distrib nat-mod-distrib* **by** *auto*
also have ... = *fst* (*?add2* (*?φ x*) (*?φ y*))
by (*auto simp*: *lucas-lehmer-hom-def lucas-lehmer-add-def case-prod-unfold*)
finally show *fst* (*?φ* (*?add1 x y*)) = *fst* (*?add2* (*?φ x*) (*?φ y*)) **.**

have *snd* (*?φ* (*?add1 x y*)) = *nat* ((*snd x* + *snd y*) *mod int m*)

by (*simp add*: *lucas-lehmer-hom-def lucas-lehmer-add'-def case-prod-unfold*)
also have (*snd x* + *snd y*) *mod int m* = ((*snd x mod m*) + (*snd y mod m*)) *mod int m*
by (*simp add*: *mod-add-eq*)
also have *nat* ... = (*nat* (*snd x mod int m*) + *nat* (*snd y mod int m*)) *mod m*
using *m nat-add-distrib nat-mod-distrib* **by** *auto*
also have ... = *snd* (*?add2* (*?φ x*) (*?φ y*))
by (*auto simp*: *lucas-lehmer-hom-def lucas-lehmer-add-def case-prod-unfold*)
finally show *snd* (*?φ* (*?add1 x y*)) = *snd* (*?add2* (*?φ x*) (*?φ y*)) **.**
qed

lemma *lucas-lehmer-hom-mult*:
lucas-lehmer-hom m (*lucas-lehmer-mult' x y*) =
lucas-lehmer-mult m (*lucas-lehmer-hom m x*) (*lucas-lehmer-hom m y*)
proof (*rule prod-eqI*)
let *?mul1* = *lucas-lehmer-mult'* **and** *?mul2* = *lucas-lehmer-mult m*
let *?φ* = *lucas-lehmer-hom m*
have *fst* (*?φ* (*?mul1 x y*)) = *nat* ((*fst x* ∗ *fst y* + *3* ∗ *snd x* ∗ *snd y*) *mod int m*)
by (*simp add*: *lucas-lehmer-hom-def lucas-lehmer-mult'-def case-prod-unfold*)
also have (*fst x* ∗ *fst y* + *3* ∗ *snd x* ∗ *snd y*) *mod int m* =
((*fst x mod int m*) ∗ (*fst y mod int m*) +
3 ∗ (*snd x mod int m*) ∗ (*snd y mod int m*)) *mod m*
by (*intro congD cong-mult cong-add cong-refl*) (*auto simp*: *cong-def*)
also have ... = *int* (*nat* (((*fst x mod int m*) ∗ (*fst y mod int m*) +
3 ∗ (*snd x mod int m*) ∗ (*snd y mod int m*)) *mod m*))
using *m* **by** (*subst of-nat-nat*) *auto*
also have ... = *int* (*nat* (*fst x mod int m*) ∗ *nat* (*fst y mod int m*) +
3 ∗ (*nat* (*snd x mod int m*)) ∗ *nat* (*snd y mod int m*)) *mod m*
using *m* **by** *simp*
also have *nat* ... = (*nat* (*fst x mod int m*) ∗ *nat* (*fst y mod int m*) +
3 ∗ *nat* (*snd x mod int m*) ∗ *nat* (*snd y mod int m*)) *mod m*
using *m* **by** (*metis nat-int zmod-int*)
also have ... = *fst* (*?mul2* (*?φ x*) (*?φ y*))
by (*simp add*: *lucas-lehmer-hom-def lucas-lehmer-mult-def case-prod-unfold*)
finally show *fst* (*?φ* (*?mul1 x y*)) = *fst* (*?mul2* (*?φ x*) (*?φ y*)) **.**

have *snd* (*?φ* (*?mul1 x y*)) = *nat* ((*fst x* ∗ *snd y* + *snd x* ∗ *fst y*) *mod int m*)
by (*simp add*: *lucas-lehmer-hom-def lucas-lehmer-mult'-def case-prod-unfold*)
also have (*fst x* ∗ *snd y* + *snd x* ∗ *fst y*) *mod int m* =
((*fst x mod int m*) ∗ (*snd y mod int m*) +
(*snd x mod int m*) ∗ (*fst y mod int m*)) *mod m*
by (*intro congD cong-mult cong-add cong-refl*) (*auto simp*: *cong-def*)
also have ... = *int* (*nat* (((*fst x mod int m*) ∗ (*snd y mod int m*) +
(*snd x mod int m*) ∗ (*fst y mod int m*)) *mod m*))
using *m* **by** (*subst of-nat-nat*) *auto*
also have ... = *int* (*nat* (*fst x mod int m*) ∗ *nat* (*snd y mod int m*) +
(*nat* (*snd x mod int m*)) ∗ *nat* (*fst y mod int m*)) *mod m*
using *m* **by** *simp*
also have *nat* ... = (*nat* (*fst x mod int m*) ∗ *nat* (*snd y mod int m*) +

```
        nat (snd x mod int m) * nat (fst y mod int m)) mod m
    using m by (metis nat-int zmod-int)
  also have ... = snd (?mul2 (?φ x) (?φ y))
    by (simp add: lucas-lehmer-hom-def lucas-lehmer-mult-def case-prod-unfold)
  finally show snd (?φ (?mul1 x y)) = snd (?mul2 (?φ x) (?φ y)) .
qed

lemma lucas-lehmer-hom-1 [simp]: lucas-lehmer-hom m (1, 0) = (1, 0)
  using m by (simp add: lucas-lehmer-hom-def)

lemma ring-hom-lucas-lehmer-hom:
  lucas-lehmer-hom m ∈ ring-hom lucas-lehmer-ring (lucas-lehmer-ring-mod m)
proof −
  interpret R: cring lucas-lehmer-ring ..
  from m interpret S: cring lucas-lehmer-ring-mod m ..
  show ?thesis
    unfolding ring-hom-def using lucas-lehmer-hom-in-carrier m
    by (auto simp: lucas-lehmer-ring-mod-def lucas-lehmer-hom-add
                   lucas-lehmer-ring-def lucas-lehmer-hom-mult)
qed

end
```

2.7 Correctness of the Lucas–Lehmer test

In this section, we will prove that the Lucas–Lehmer test is both a necessary and sufficient condition for the primality of a Mersenne number of the form $2^p - 1$ for an odd prime p. The proof that shall be given here is rather explicit and heavily draws from the Wikipedia article on the Lucas–Lehmer test [3].

A shorter and more high-level proof of a more general statement can be obtained using more theory on finite fields (in particular the field $\mathrm{GF}(q^2)$ (cf. e. g. Rödseth [2]).

```
definition lucas-lehmer-test where
  lucas-lehmer-test p = (p > 2 ∧
    (2 ^ p − 1) dvd gen-lucas-lehmer-sequence 4 (p − 2))
```

We can now prove that any Mersenne number $2^p - 1$ for p prime that passes the Lucas–Lehmer test is prime. We follow the simple argument given by Bruce [1], which is also given on Wikipedia [3].

```
theorem lucas-lehmer-sufficient:
  assumes prime p odd p
  assumes (2 ^ p − 1) dvd gen-lucas-lehmer-sequence 4 (p − 2)
  shows   prime (2 ^ p − 1 :: nat)
proof (rule ccontr)
  assume not-prime: ¬prime (2 ^ p − 1 :: nat)
```

```
  from assms obtain k :: int where k: gen-lucas-lehmer-sequence 4 (p − 2) = k
* (2 ^ p − 1)
    by (elim dvdE) (auto simp: mult-ac)
  from assms have p > 2
    using odd-prime-gt-2-nat by blast
  from ‹p > 2› have 2 ^ p ≥ (2 ^ 3 :: nat) by (intro power-increasing) auto
  hence 2 ^ p ≥ (8 :: nat) by simp

  define q :: nat where q = Min (prime-factors (2 ^ p − 1))
  have q ∈ prime-factors (2 ^ p − 1) using ‹2 ^ p ≥ 8›
    unfolding q-def by (intro Min-in) (auto simp: prime-factorization-empty-iff)
  hence q: prime q q dvd (2 ^ p − 1 :: nat)
    by (auto simp: in-prime-factors-iff)
  have q-minimal: q ≤ q' if q' ∈ prime-factors (2 ^ p − 1) for q'
    unfolding q-def by (rule Min-le) (use that in auto)

  have 2 ^ p − 1 ≥ q ^ 2
  proof −
    from q obtain k where k: 2 ^ p − 1 = q * k by auto
    have prime-factorization (2 ^ p − 1 :: nat) ≠ {#q#}
    proof
      assume *: prime-factorization (2 ^ p − 1 :: nat) = {#q#}
      have 2 ^ p − 1 = prod-mset (prime-factorization (2 ^ p − 1 :: nat))
        using ‹2 ^ p ≥ 8› by (subst prod-mset-prime-factorization-nat) auto
      also have ... = q by (subst *) auto
      finally show False using not-prime q by simp
    qed
    hence prime-factorization k ≠ {#} using q k ‹2 ^ p ≥ 8›
      by (subst (asm) k, subst (asm) prime-factorization-mult)
        (auto intro!: Nat.gr0I simp: prime-factorization-prime)
    hence k ≠ 1 by (auto simp: prime-factorization-empty-iff)
    then obtain q' where q': prime q' q' dvd k
      using prime-factor-nat by blast
    from q' k ‹2 ^ p ≥ 8› have q ≤ q'
      by (intro q-minimal) (auto simp: in-prime-factors-iff intro!: Nat.gr0I)
    hence q ^ 2 ≤ q * q'
      unfolding power2-eq-square by (intro mult-mono) auto
    also have q * q' ≤ 2 ^ p − 1
      using q q' k ‹2 ^ p ≥ 8› by (intro dvd-imp-le) (auto intro!: Nat.gr0I)
    finally show 2 ^ p − 1 ≥ q ^ 2 .
  qed

  have q ≠ 2 using q ‹p > 2› by auto
  moreover from q have q ≠ 0 q ≠ 1 by auto
  ultimately have q > 2 by auto

  write lucas-lehmer-ring (R)
  define S where S = lucas-lehmer-ring-mod q
  define S' where S' = units-of S
```

define φ **where** φ = *lucas-lehmer-hom q*

interpret *R*: *cring R* **..**
interpret *S*: *cring S*
unfolding *S-def* **by** (*rule cring-lucas-lehmer-ring-mod*) (*use* ‹*q* > *2*› **in** *auto*)
interpret *S'*: *comm-group S'*
unfolding *S'-def* **by** (*rule S.units-comm-group*)
have $\varphi \in$ *ring-hom R S*
unfolding φ*-def S-def* **by** (*rule ring-hom-lucas-lehmer-hom*) (*use* ‹*q* > *2*› **in** *auto*)
interpret φ: *ring-hom-cring R S* φ
by *standard fact*

have (*2* + *sqrt 3*) ^ (*2* ^ (*p* − *2*)) + (*2* − *sqrt 3*) ^ (*2* ^ (*p* − *2*)) =
real-of-int (*gen-lucas-lehmer-sequence 4* (*p* − *2*))
unfolding *gen-lucas-lehmer-sequence-4-closed-form1* **..**
also have ... = *real-of-int k* ∗ (*2* ^ *p* − *1*)
by (*simp add*: *k*)
finally have ∗: (*2* + *sqrt 3*) ^ (*2* ^ (*p* − *2*)) =
real-of-int k ∗ (*2* ^ *p* − *1*) − (*2* − *sqrt 3*) ^ (*2* ^ (*p* − *2*))
by (*simp add*: *algebra-simps*)
have ((*2* + *sqrt 3*) ^ (*2* ^ (*p* − *2*))) ^ *2* =
real-of-int k ∗ (*2* ^ *p* − *1*) ∗ (*2* + *sqrt 3*) ^ (*2* ^ (*p* − *2*)) −
(*2* − *sqrt 3*) ^ (*2* ^ (*p* − *2*)) ∗ (*2* + *sqrt 3*) ^ (*2* ^ (*p* − *2*))
unfolding *power2-eq-square* **by** (*subst* ∗) (*simp add*: *algebra-simps*)
also have ((*2* + *sqrt 3*) ^ (*2* ^ (*p* − *2*))) ^ *2* = (*2* + *sqrt 3*) ^ (*2* ∗ *2* ^ (*p* − *2*))
by (*simp flip*: *power-mult add*: *mult-ac*)
also have *2* ∗ *2* ^ (*p* − *2*) = *2* ^ (*Suc* (*p* − *2*))
by *simp*
also from ‹*p* > *2*› **have** *Suc* (*p* − *2*) = *p* − *1*
by *linarith*
also have (*2* − *sqrt 3*) ^ (*2* ^ (*p* − *2*)) ∗ (*2* + *sqrt 3*) ^ (*2* ^ (*p* − *2*)) = *1*
by (*subst power-mult-distrib* [*symmetric*]) (*auto simp*: *algebra-simps*)
finally have (*2* + *sqrt 3*) ^ (*2* ^ (*p* − *1*)) =
real-of-int k ∗ (*2* ^ *p* − *1*) ∗ (*2* + *sqrt 3*) ^ (*2* ^ (*p* − *2*)) − *1* **.**

also have (*2* + *sqrt 3*) ^ (*2* ^ (*p* − *1*)) =
lucas-lehmer-to-real ((*2*, *1*) [^]$_R$ (*2* ^ (*p* − *1*) :: *nat*))
by (*simp add*: *lucas-lehmer-to-real-simps*)
also have *real-of-int k* ∗ (*2* ^ *p* − *1*) ∗ (*2* + *sqrt 3*) ^ (*2* ^ (*p* − *2*)) − *1* =
lucas-lehmer-to-real ((*k* ∗ (*2* ^ *p* − *1*), *0*) ⊗$_R$
(*2*, *1*) [^]$_R$ (*2* ^ (*p* − *2*) :: *nat*) ⊕$_R$ ⊖$_R$ **1**$_R$)
by (*simp add*: *lucas-lehmer-to-real-simps*)
finally have ((*2*, *1*) [^]$_R$ (*2* ^ (*p* − *1*) :: *nat*)) =
((*k* ∗ (*2* ^ *p* − *1*), *0*) ⊗$_R$ (*2*, *1*) [^]$_R$ (*2* ^ (*p* − *2*) :: *nat*) ⊕$_R$ ⊖$_R$ **1**$_R$)
by (*rule injD*[*OF lucas-lehmer-to-real-inj*])

hence φ ((*2*, *1*) [^]$_R$ (*2* ^ (*p* − *1*) :: *nat*)) =

```
        φ ((k * (2 ^ p - 1), 0) ⊗_R (2, 1) [^]_R (2 ^ (p - 2) :: nat) ⊕_R ⊖_R 1_R)
    by (simp only:)
  also have φ ((2, 1) [^]_R (2 ^ (p - 1) :: nat)) = φ (2, 1) [^]_S (2 ^ (p - 1) ::
nat)
    by simp
  also {
    have int q dvd int (2 ^ p - 1)
      by (subst int-dvd-int-iff) (use q in auto)
    also have int (2 ^ p - 1) = 2 ^ p - 1
      by (simp add: of-nat-diff)
    finally have φ (k * (2 ^ p - 1), 0) = 0_S
      by (simp add: φ-def lucas-lehmer-hom-def S-def lucas-lehmer-ring-mod-def)
  }
  hence φ ((k * (2 ^ p - 1), 0) ⊗_R (2, 1) [^]_R (2 ^ (p - 2) :: nat) ⊕_R ⊖_R 1_R)
=
        ⊖_S 1_S
    by simp
  finally have eq: φ (2, 1) [^]_S (2 ^ (p - 1) :: nat) = ⊖_S 1_S .

  have φ (2, 1) [^]_S (2 ^ p :: nat) = φ (2, 1) [^]_S (2 ^ (p - 1) * 2 :: nat)
    using ‹p > 2› by (cases p) (auto simp: mult-ac)
  also have ... = (φ (2, 1) [^]_S (2 ^ (p - 1) :: nat)) [^]_S (2 :: nat)
    by (subst S.nat-pow-pow) auto
  also have ... = 1_S
    by (subst eq) (auto simp: numeral-2-eq-2 S.l-minus)
  finally have eq': φ (2, 1) [^]_S (2 ^ p :: nat) = 1_S .

  from eq' have unit: φ (2, 1) ∈ Units S
    by (rule S.pow-nat-eq-1-imp-unit) auto

  have neg-one-not-one: ⊖_S 1_S ≠ 1_S
  proof
    assume *: ⊖_S 1_S = 1_S
    have (⊖_S 1_S) ⊕_S 1_S = 0_S
      by (rule S.l-neg) auto
    hence 1_S ⊕_S 1_S = 0_S
      by (simp only: *)
    thus False using ‹q > 2›
      by (auto simp: S-def lucas-lehmer-ring-mod-def lucas-lehmer-add-def)
  qed

  have fin: finite (Units S)
   by (rule finite-subset[of - carrier S]) (auto simp: Units-def S-def lucas-lehmer-ring-mod-def)

  have group.ord S' (φ (2, 1)) = 2 ^ p
    using ‹p > 2› eq eq' unit neg-one-not-one
    by (intro S'.ord-eqI-prime-factors)
       (auto simp: prime-factors-power prime-factorization-prime
                   S'-def S.units-of-pow units-of-carrier units-of-one power-diff)
```

```
  hence 2 ^ p = group.ord S' (φ (2, 1))
    by simp
  also have ... = card (generate S' {φ (2, 1)})
    using unit fin
    by (intro S'.generate-pow-card) (auto simp: S'-def units-of-carrier)
  also have ... ≤ card (carrier S')
   using fin unit by (intro card-mono S'.generate-incl) (auto simp: S'-def units-of-carrier)
  also have ... < q ^ 2
    unfolding S'-def S-def using card-lucas-lehmer-Units[of q] ‹q > 2›
    by (auto simp: units-of-carrier)
  also note ‹q ^ 2 ≤ 2 ^ p - 1›
  finally show False by simp
qed
```

Next, we show that any Mersenne prime passes the Lucas–Lehmer test. We again follow the rather explicit proof outlined on Wikipedia [3], which is a simplified (but less general and less abstract) version of the proof by Rödseth [2].

```
theorem (in mersenne-prime) lucas-lehmer-necessary:
  (2 ^ p - 1) dvd gen-lucas-lehmer-sequence 4 (p - 2)
proof -
  write lucas-lehmer-ring (R)
  define S where S = lucas-lehmer-ring-mod M
  define S' where S' = units-of S
  define φ where φ = lucas-lehmer-hom M

  interpret R: cring R ..
  interpret S: cring S unfolding S-def
    by (rule cring-lucas-lehmer-ring-mod) (use M-gt-6 in auto)
  interpret S': comm-group S'
    unfolding S'-def by (rule S.units-comm-group)
  have φ ∈ ring-hom R S unfolding φ-def S-def
    by (rule ring-hom-lucas-lehmer-hom) (use M-gt-6 in auto)
  interpret φ: ring-hom-cring R S φ
    by standard fact

  have R-pow-int: (n, 0) [^]R m = (n ^ m, 0) for n :: int and m :: nat
    by (induction m; simp; simp add: lucas-lehmer-ring-def lucas-lehmer-mult'-def)

  have add-pow R n 1R = (int n, 0) for n
    by (induction n; simp; simp add: lucas-lehmer-ring-def lucas-lehmer-add'-def)
  hence add-pow R M 1R = (int M, 0)
    by simp
  also have φ ... = 0S
    by (simp add: φ-def S-def lucas-lehmer-ring-mod-def lucas-lehmer-hom-def)
  finally have add-pow S M 1S = 0S
    by (simp add: φ.hom-add-pow-nat)

  define σ :: int × int where σ = (0, 2)
```

have *eq1*: $\varphi\ ((6,\ 2)\ [\widehat{}]_R\ M) = \varphi\ (6,\ -2)$
proof −
have $(6,\ 2) = (6,\ 0) \oplus_R \sigma$
by (*simp add*: *lucas-lehmer-ring-def* σ*-def lucas-lehmer-add′-def*)
also have $\varphi\ (\ldots\ [\widehat{}]_R\ M) = \varphi\ ((6,\ 0)\ [\widehat{}]_R\ M) \oplus_S \varphi\ (\sigma\ [\widehat{}]_R\ M)$
using *prime* **and** ‹*add-pow S M* $\mathbf{1}_S = \mathbf{0}_S$›
by (*simp add*: *S.binomial-finite-char*)
also have $(6,\ 0)\ [\widehat{}]_R\ M = (6\ \widehat{}\ M,\ 0)$
by (*simp add*: *R-pow-int*)
also have $[6\ \widehat{}\ M = 6]\ (mod\ (int\ M))$ **using** *M-gt-6*
by (*intro little-Fermat-int*) (*use prime* **in** ‹*auto simp flip*: *dvd-nat-abs-iff*›)
hence $\varphi\ (6\ \widehat{}\ M,\ 0) = \varphi\ (6,\ 0)$
unfolding φ*-def* **by** (*intro lucas-lehmer-hom-cong*) *auto*
also have $\sigma = (2,\ 0) \otimes_R (0,\ 1)$
by (*simp add*: σ*-def lucas-lehmer-ring-def lucas-lehmer-mult′-def*)
hence $\varphi\ (\sigma\ [\widehat{}]_R\ M) = \varphi\ ((2,\ 0)\ [\widehat{}]_R\ M \otimes_R (0,\ 1)\ [\widehat{}]_R\ M)$
by (*subst R.nat-pow-distrib* [*symmetric*]) *auto*
also have $\ldots = \varphi\ ((2,\ 0)\ [\widehat{}]_R\ M) \otimes_S \varphi\ ((0,\ 1)\ [\widehat{}]_R\ M)$
by *simp*
also have $(2,\ 0)\ [\widehat{}]_R\ M = (2\ \widehat{}\ M,\ 0)$
by (*simp add*: *R-pow-int*)
also have $[2\ \widehat{}\ M = 2]\ (mod\ int\ M)$ **using** *M-gt-6 prime*
by (*intro little-Fermat-int*) (*auto simp flip*: *dvd-nat-abs-iff dest*: *dvd-imp-le*)
hence $\varphi\ (2\ \widehat{}\ M,\ 0) = \varphi\ (2,\ 0)$
unfolding φ*-def* **by** (*intro lucas-lehmer-hom-cong*) *auto*
also have *M-eq*: $M = Suc\ (2 * ((M - 1)\ div\ 2))$
using *M-odd* **by** *auto*
have $(0,\ 1)\ [\widehat{}]_R\ M = (0,\ 1) \otimes_R ((0,\ 1)\ [\widehat{}]_R\ (2{::}nat))\ [\widehat{}]_R\ ((M - 1)\ div\ 2)$
by (*subst M-eq*) (*auto simp*: *R.nat-pow-mult R.nat-pow-pow R.cring-simprules*)
also have $(0,\ 1)\ [\widehat{}]_R\ (2{::}nat) = (3,\ 0)$
by (*simp add*: *eval-nat-numeral*) (*simp add*: *lucas-lehmer-ring-def lucas-lehmer-mult′-def*)
also have $\varphi\ ((0,\ 1) \otimes_R (3,\ 0)\ [\widehat{}]_R\ ((M - 1)\ div\ 2)) =$
$\varphi\ ((3,\ 0)\ [\widehat{}]_R\ ((M - 1)\ div\ 2)) \otimes_S \varphi\ (0,\ 1)$
by (*simp add*: *S.cring-simprules*)
also have $(3,\ 0)\ [\widehat{}]_R\ ((M - 1)\ div\ 2) = (3\ \widehat{}\ ((M - 1)\ div\ 2),\ 0)$
by (*simp add*: *R-pow-int*)
also have $\varphi\ (3\ \widehat{}\ ((M - 1)\ div\ 2),\ 0) = \varphi\ (-1,\ 0)$
unfolding φ*-def*
proof (*intro lucas-lehmer-hom-cong′*)
have $[3\ \widehat{}\ ((M - 1)\ div\ 2) = Legendre\ 3\ M]\ (mod\ int\ M)$
by (*rule cong-sym*, *rule euler-criterion*) (*use prime M-gt-6* **in** *auto*)
thus $[3\ \widehat{}\ ((M - 1)\ div\ 2) = -1]\ (mod\ int\ M)$
by (*simp add*: *Legendre-3-M*)
qed *auto*
also have $\varphi\ (2,\ 0) \otimes_S (\varphi\ (-\ 1,\ 0) \otimes_S \varphi\ (0,\ 1)) = \varphi\ ((2,\ 0) \otimes_R (-1,\ 0) \otimes_R (0,\ 1))$
by (*simp add*: *R.cring-simprules S.cring-simprules*)
also have $\varphi\ (6,\ 0) \oplus_S \varphi\ ((2,\ 0) \otimes_R (-\ 1,\ 0) \otimes_R (0,\ 1)) =$
$\varphi\ ((6,\ 0) \oplus_R (2,\ 0) \otimes_R (-\ 1,\ 0) \otimes_R (0,\ 1))$

```
    by simp
  also have ... = φ (6, −2) unfolding φ-def
    by (intro lucas-lehmer-hom-cong)
     (auto simp: lucas-lehmer-ring-def lucas-lehmer-mult'-def lucas-lehmer-add'-def)
  finally show φ ((6, 2) [^]_R M) = φ (6, −2)
    by (simp add: R.cring-simprules S.cring-simprules)
qed

have eq2: φ ((24, 0) [^]_R ((M − 1) div 2)) = ⊖_S 1_S
proof −
  have (24, 0) = (2, 0) [^]_R (3::nat) ⊗_R (3, 0)
  by (simp add: eval-nat-numeral) (auto simp: lucas-lehmer-ring-def lucas-lehmer-mult'-def)
  also have ... [^]_R ((M − 1) div 2) =
            ((2, 0) [^]_R ((M − 1) div 2)) [^]_R (3::nat) ⊗_R (3, 0) [^]_R ((M −
1) div 2)
    by (simp add: R.cring-simprules R.nat-pow-distrib R.nat-pow-pow mult-ac)
  also have φ ... = (φ ((2, 0) [^]_R ((M − 1) div 2))) [^]_S (3::nat) ⊗_S
                φ ((3, 0) [^]_R ((M − 1) div 2)) by simp
  also have (2, 0) [^]_R ((M − 1) div 2) = (2 ^ ((M − 1) div 2), 0)
    by (simp add: R-pow-int)
  also have φ ... = φ (1, 0)
    unfolding φ-def
  proof (intro lucas-lehmer-hom-cong')
    have [2 ^ ((M − 1) div 2) = Legendre 2 M] (mod int M)
      by (rule cong-sym, rule euler-criterion) (use prime M-gt-6 in auto)
    thus [2 ^ ((M − 1) div 2) = 1] (mod int M)
      using Legendre-2-M by simp
  qed auto
  also have (1, 0) = 1_R
    by (simp add: lucas-lehmer-ring-def)
  also have (3, 0) [^]_R ((M − 1) div 2) = (3 ^ ((M − 1) div 2), 0)
    by (simp add: R-pow-int)
  also have φ ... = φ (−1, 0)
    unfolding φ-def
  proof  (intro lucas-lehmer-hom-cong')
    have [3 ^ ((M − 1) div 2) = Legendre 3 M] (mod int M)
      by (rule cong-sym, rule euler-criterion) (use prime M-gt-6 in auto)
    thus [3 ^ ((M − 1) div 2) = −1] (mod int M)
      using Legendre-3-M by simp
  qed auto
  also have (−1, 0) = ⊖_R 1_R
    using minus-lucas-lehmer-ring by (simp add: lucas-lehmer-ring-def)
  finally show φ ((24, 0) [^]_R ((M − 1) div 2)) = ⊖_S 1_S
    by simp
qed

define ω ω' :: int × int where ω = (2, 1) and ω' = (2, −1)
have eq3: φ (ω [^]_R ((M + 1) div 2)) = ⊖_S 1_S
proof −
```

```
  have (M + 1) div 2 = Suc ((M − 1) div 2)
    using M-odd M-gt-6 by (auto elim!: oddE)
  have ∗: φ ((24, 0) ⊗R ω) = φ ((6, 2) [^]R (2 :: nat)) unfolding φ-def
    by (intro lucas-lehmer-hom-cong)
       (simp-all add: eval-nat-numeral,
        auto simp: lucas-lehmer-ring-def lucas-lehmer-mult'-def ω-def)
  have φ (⊖R 1R) ⊗S φ ((24, 0) ⊗R ω) [^]S ((M + 1) div 2) =
         φ (⊖R 1R) ⊗S φ ((6, 2) [^]R (2 :: nat)) [^]S ((M + 1) div 2)
    by (subst ∗) auto
  hence φ (⊖R 1R) ⊗S φ ((24, 0) [^]R ((M + 1) div 2)) ⊗S φ (ω [^]R ((M +
1) div 2)) =
         φ (⊖R 1R) ⊗S φ ((6, 2) [^]R (2 ∗ ((M + 1) div 2)))
    by (simp add: R.nat-pow-distrib S.nat-pow-distrib R.nat-pow-pow
                  S.nat-pow-pow R.cring-simprules S.cring-simprules)
  also have 2 ∗ ((M + 1) div 2) = M + 1
    using M-odd by auto
  finally have φ (24, 0) ⊗S (φ (⊖R 1R) ⊗S φ ((24, 0) [^]R ((M − 1) div 2)))
⊗S
                 φ (ω [^]R ((M + 1) div 2)) =
               φ (⊖R 1R) ⊗S (φ (6, 2) ⊗S φ ((6, 2) [^]R M))
    by (subst (asm) ‹(M + 1) div 2 = -›) (simp add: S.cring-simprules R.cring-simprules)

  also have φ ((24, 0) [^]R ((M − 1) div 2)) = ⊖S 1S
    by (subst eq2) auto
  also have (φ (⊖R 1R) ⊗S ⊖S 1S) = 1S
    by (simp add: S.cring-simprules)
  also have φ ((6, 2) [^]R M) = φ (6, −2)
    by (subst eq1) auto
  also have φ (6, 2) ⊗S φ (6, −2) = φ ((6, 2) ⊗R (6, −2))
    by simp
  also have ... = φ (24, 0) unfolding φ-def
    by (intro lucas-lehmer-hom-cong) (auto simp: lucas-lehmer-ring-def lucas-lehmer-mult'-def)
  finally have φ (24, 0) ⊗S (φ (ω [^]R ((M + 1) div 2))) =
               φ (24, 0) ⊗S φ (⊖R 1R)
    by (simp add: S.cring-simprules)
  also have φ (24, 0) = (24 mod M, 0)
    by (simp add: φ-def lucas-lehmer-hom-def nat-mod-as-int)
  finally have (24 mod M, 0) ⊗S (φ (ω [^]R ((M + 1) div 2))) =
               (24 mod M, 0) ⊗S φ (⊖R 1R) .
  moreover have (24 mod M, 0) ∈ Units S
    unfolding S-def using M-gt-6 prime M-not-dvd-24
    by (intro int-in-Units-lucas-lehmer-ring-mod) (auto simp: dvd-mod-iff intro!:
Nat.gr0I)
  ultimately show φ (ω [^]R ((M + 1) div 2)) = ⊖S 1S
    by (subst (asm) S.Units-l-cancel) auto
qed

have eq4: φ (ω [^]R (2 ^ (p − 2) :: nat) ⊕R ω' [^]R (2 ^ (p − 2) :: nat)) = 0S
  (is φ ?lhs = -)
```

```
  proof −
    have φ (ω [^]_R ((M + 1) div 2)) ⊗_S φ (ω' [^]_R ((M + 1) div 4)) ⊕_S
          φ (ω' [^]_R ((M + 1) div 4)) = 0_S
      by (subst eq3) (auto simp: S.cring-simprules)
    also have 2 ^ 2 dvd (2 ^ p :: nat)
      by (intro le-imp-power-dvd) (use p-gt-2 in auto)
    hence 4 dvd (M + 1) by (auto simp: M-def)
    hence (M + 1) div 2 = (M + 1) div 4 + (M + 1) div 4
      by presburger
    also have φ (ω [^]_R ...) ⊗_S φ (ω' [^]_R ((M + 1) div 4)) =
            φ (ω ⊗_R ω') [^]_S ((M + 1) div 4) ⊗_S φ (ω [^]_R ((M + 1) div 4))
      by (simp add: S.cring-simprules S.nat-pow-distrib flip: S.nat-pow-mult)
    also have φ (ω ⊗_R ω') = φ 1_R unfolding φ-def
      by (intro lucas-lehmer-hom-cong)
        (auto simp: ω-def ω'-def lucas-lehmer-ring-def lucas-lehmer-mult'-def)
    also have (M + 1) div 4 = 2 ^ (p − 2)
      using p-gt-2 by (auto simp: M-def power-diff)
    finally show eq4: φ (ω [^]_R (2 ^ (p − 2) :: nat) ⊕_R ω' [^]_R (2 ^ (p − 2) ::
nat)) = 0_S
      by simp
  qed

  have φ ?lhs = 0_S
    by (rule eq4)
  also have lucas-lehmer-to-real ?lhs =
            lucas-lehmer-to-real (gen-lucas-lehmer-sequence 4 (p − 2), 0)
   by (simp add: ω-def ω'-def lucas-lehmer-to-real-simps gen-lucas-lehmer-sequence-4-closed-form1)
  hence ?lhs = (gen-lucas-lehmer-sequence 4 (p − 2), 0)
    by (rule injD[OF lucas-lehmer-to-real-inj])
  finally have gen-lucas-lehmer-sequence 4 (p − 2) mod M = 0 using M-gt-6
    by (auto simp: φ-def lucas-lehmer-hom-def S-def lucas-lehmer-ring-mod-def)
  thus (2 ^ p − 1) dvd gen-lucas-lehmer-sequence 4 (p − 2)
    by (simp add: M-def mod-eq-0-iff-dvd of-nat-diff)
qed

corollary lucas-lehmer-correct:
  prime (2 ^ p − 1 :: nat) ⟷
     prime p ∧ (p = 2 ∨ (2 ^ p − 1) dvd gen-lucas-lehmer-sequence 4 (p − 2))
proof (intro iffI; (elim conjE)?)
  assume prime: prime (2 ^ p − 1 :: nat)
  from prime have p ≠ 0 p ≠ 1
    by (auto intro!: Nat.gr0I)
  hence p = 2 ∨ p > 2 by auto
  thus prime p ∧ (p = 2 ∨ (2 ^ p − 1) dvd gen-lucas-lehmer-sequence 4 (p − 2))
  proof (elim disjE)
    assume p > 2
    with prime interpret mersenne-prime p 2 ^ p − 1
      by unfold-locales
    from lucas-lehmer-necessary p-prime show ?thesis by auto
```

```
  qed auto
next
  assume prime: prime p and *: p = 2 ∨ (2 ^ p − 1) dvd gen-lucas-lehmer-sequence
4 (p − 2)
  from * consider p = 2 | p ≠ 2 (2 ^ p − 1) dvd gen-lucas-lehmer-sequence 4 (p
− 2)
    by auto
  thus prime (2 ^ p − 1 :: nat)
  proof cases
    assume p ≠ 2 and dvd: (2 ^ p − 1) dvd gen-lucas-lehmer-sequence 4 (p − 2)
    from ‹prime p› and ‹p ≠ 2› have p > 2
      using prime-gt-1-nat[of p] by auto
    with prime have odd p by (auto simp: prime-odd-nat)
    with prime dvd show ?thesis
      by (intro lucas-lehmer-sufficient)
  qed auto
qed

corollary lucas-lehmer-correct':
  prime (2 ^ p − 1 :: nat) ⟷ prime p ∧ (p = 2 ∨ lucas-lehmer-test p)
  using lucas-lehmer-correct[of p] prime-gt-1-nat[of p]
  by (auto simp: lucas-lehmer-test-def)
```

2.8 A first executable version Lucas–Lehmer test

The following is an implementation of the Lucas–Lehmer test using modular arithmetic on the integers. This is not the most efficient implementation – the modular arithmetic can be replaced by much cheaper bitwise operations, and we will do that in the next section.

```
primrec gen-lucas-lehmer-sequence' :: int ⇒ int ⇒ nat ⇒ int where
  gen-lucas-lehmer-sequence' m a 0 = a
| gen-lucas-lehmer-sequence' m a (Suc n) = gen-lucas-lehmer-sequence' m ((a ^ 2
− 2) mod m) n

lemma gen-lucas-lehmer-sequence'-Suc':
  gen-lucas-lehmer-sequence' m a (Suc n) = (gen-lucas-lehmer-sequence' m a n ^ 2
− 2) mod m
  by (induction n arbitrary: a) auto

lemma gen-lucas-lehmer-sequence'-correct:
  assumes a ∈ {0..<m}
  shows   gen-lucas-lehmer-sequence' m a n = gen-lucas-lehmer-sequence a n mod
m
  using assms
proof (induction n)
  case (Suc n)
  have gen-lucas-lehmer-sequence' m a (Suc n) =
        ((gen-lucas-lehmer-sequence a n mod m)^2 − 2) mod m
```

using *Suc* **unfolding** *gen-lucas-lehmer-sequence'-Suc'* **by** *simp*
also have ... = ((*gen-lucas-lehmer-sequence a n*)^2 − *2*) *mod m*
by (*intro congD cong-diff cong-pow cong-refl*) (*auto simp*: *cong-def*)
finally show *?case* **by** *simp*
qed *auto*

lemma *lucas-lehmer-test-code-arithmetic* [*code*]:
lucas-lehmer-test p = (*p* > *2* ∧
gen-lucas-lehmer-sequence' (*2* ^ *p* − *1*) *4* (*p* − *2*) = *0*)
unfolding *lucas-lehmer-test-def*
proof (*intro conj-cong refl*)
assume *p*: *p* > *2*
from *p* **have** *2* ^ *p* ≥ (*2* ^ *3* :: *int*) **by** (*intro power-increasing*) *auto*
have (*2* ^ *p* − *1 dvd gen-lucas-lehmer-sequence 4* (*p* − *2*)) ⟷
gen-lucas-lehmer-sequence 4 (*p* − *2*) *mod* (*2* ^ *p* − *1*) = *0*
by *auto*
also have *gen-lucas-lehmer-sequence 4* (*p* − *2*) *mod* (*2* ^ *p* − *1*) =
gen-lucas-lehmer-sequence' (*2* ^ *p* − *1*) *4* (*p* − *2*)
using ‹*2* ^ *p* ≥ *2* ^ *3*›
by (*intro gen-lucas-lehmer-sequence'-correct* [*symmetric*]) *auto*
finally show (*2* ^ *p* − *1 dvd gen-lucas-lehmer-sequence 4* (*p* − *2*)) =
(*gen-lucas-lehmer-sequence'* (*2* ^ *p* − *1*) *4* (*p* − *2*) = *0*) .
qed

lemma *mersenne-prime-iff*: *mersenne-prime p* ⟷ *p* > *2* ∧ *prime* (*2* ^ *p* − *1* :: *nat*)
by (*simp add*: *mersenne-prime-def*)

lemma *mersenne-prime-code* [*code*]:
mersenne-prime p ⟷ *prime p* ∧ *lucas-lehmer-test p*
unfolding *mersenne-prime-iff* **using** *lucas-lehmer-correct'*[*of p*]
by (*auto simp*: *lucas-lehmer-test-def*)

end

3 Efficient code for testing Mersenne primes

theory *Lucas-Lehmer-Code*
imports
Lucas-Lehmer
HOL−Library.Code-Target-Numeral
Native-Word.Code-Target-Int-Bit
begin

3.1 Efficient computation of remainders modulo a Mersenne number

We have $k = k \bmod 2^n + k \operatorname{div} 2^n \pmod{(2^n - 1)}$, and $k \bmod 2^n = k \mathbin{\&} (2^n - 1)$ and $k \operatorname{div} 2^n = k \gg n$. Therefore, we can reduce k modulo $2^n - 1$ using only bitwise operations, addition, and bit shifts.

lemma *cong-mersenne-number-int*:
  fixes *k* :: *int*
  shows [*k mod 2 ^ n + k div 2 ^ n = k*] (*mod* (*2 ^ n − 1*))
proof −
  have *k* = (*2 ^ n − 1 + 1*) ∗ (*k div 2 ^ n*) + (*k mod 2 ^ n*)
    by *simp*
  also have [. . . = (*0 + 1*) ∗ (*k div 2 ^ n*) + (*k mod 2 ^ n*)] (*mod* (*2 ^ n − 1*))
    by (*intro cong-add cong-mult cong-refl*) (*auto simp*: *cong-def*)
  finally show *?thesis* **by** (*simp add*: *cong-sym add-ac*)
qed

We encapsulate a single reduction step in the following operation. Note, however, that the result is not, in general, the same as $k \bmod (2^n - 1)$. Multiple reductions might be required in order to reduce it below 2^n, and a multiple of $2^n - 1$ can be reduced to $2^n - 1$, which is invariant to further reduction steps.

definition *mersenne-mod* :: *int* ⇒ *nat* ⇒ *int* **where**
  mersenne-mod k n = *k mod 2 ^ n + k div 2 ^ n*

lemma *mersenne-mod-code* [*code*]:
  mersenne-mod k n = *take-bit n k* + *drop-bit n k*
  by (*simp add*: *mersenne-mod-def flip*: *take-bit-eq-mod drop-bit-eq-div*)

lemma *cong-mersenne-mod*: [*mersenne-mod k n* = *k*] (*mod* (*2 ^ n − 1*))
  unfolding *mersenne-mod-def* **by** (*rule cong-mersenne-number-int*)

lemma *mersenne-mod-nonneg* [*simp*]: *k* ≥ *0* ⟹ *mersenne-mod k n* ≥ *0*
 unfolding *mersenne-mod-def* **by** (*intro add-nonneg-nonneg*) (*simp-all add*: *pos-imp-zdiv-nonneg-iff*)

lemma *mersenne-mod-less*:
  assumes *k* ≤ *2 ^ m m* ≥ *n*
  shows *mersenne-mod k n* < *2 ^ n + 2 ^* (*m − n*)
proof −
  have *mersenne-mod k n* = *k mod 2 ^ n + k div 2 ^ n*
    by (*simp add*: *mersenne-mod-def*)
  also have *k mod 2 ^ n* < *2 ^ n*
    by *simp*
  also {
    have *k div 2 ^ n ∗ 2 ^ n + 0* ≤ *k div 2 ^ n ∗ 2 ^ n + k mod* (*2 ^ n*)
      by (*intro add-mono*) *auto*
    also have . . . = *k*
      by (*subst mult.commute*) *auto*

```
    also have ... ≤ 2 ^ m
      using assms by simp
    also have ... = 2 ^ (m − n) * 2 ^ n
      using assms by (simp flip: power-add)
    finally have k div 2 ^ n ≤ 2 ^ (m − n)
      by simp
  }
  finally show ?thesis by simp
qed

lemma mersenne-mod-less':
  assumes k ≤ 5 * 2 ^ n
  shows   mersenne-mod k n < 2 ^ n + 5
proof −
  have mersenne-mod k n = k mod 2 ^ n + k div 2 ^ n
    by (simp add: mersenne-mod-def)
  also have k mod 2 ^ n < 2 ^ n
    by simp
  also {
    have k div 2 ^ n * 2 ^ n + 0 ≤ k div 2 ^ n * 2 ^ n + k mod (2 ^ n)
      by (intro add-mono) auto
    also have ... = k
      by (subst mult.commute) auto
    also have ... ≤ 5 * 2 ^ n
      using assms by simp
    finally have k div 2 ^ n ≤ 5
      by simp
  }
  finally show ?thesis by simp
qed
```

It turns out that for our use case, a single reduction is not enough to reduce the number in question enough (or at least I was unable to prove that it is). We therefore perform two reduction steps, which is enough to guarantee that our numbers are below $2^n + 4$ before and after every step in the Lucas–Lehmer sequence.

Whether one or two reductions are performed is not very important anyway, since the dominant step is the squaring anyway.

```
definition mersenne-mod2 :: int ⇒ nat ⇒ int where
  mersenne-mod2 k n = mersenne-mod (mersenne-mod k n) n

lemma cong-mersenne-mod2: [mersenne-mod2 k n = k] (mod (2 ^ n − 1))
  unfolding mersenne-mod2-def by (rule cong-trans) (rule cong-mersenne-mod)+

lemma mersenne-mod2-nonneg [simp]: k ≥ 0 ⟹ mersenne-mod2 k n ≥ 0
  unfolding mersenne-mod2-def by simp

lemma mersenne-mod2-less:
```

assumes $n > 2$ **and** $k \leq 2 \hat{} (2 * n + 2)$
shows *mersenne-mod2 k n* $< 2 \hat{} n + 5$
proof −
from *assms* **have** $2 \hat{} 3 \leq (2 \hat{} n :: int)$
by (*intro power-increasing*) *auto*
hence $2 \hat{} n \geq (8 :: int)$ **by** *simp*
have *mersenne-mod k n* $< 2 \hat{} n + 2 \hat{} (2 * n + 2 - n)$
by (*rule mersenne-mod-less*) (*use assms* **in** *auto*)
also have $\ldots \leq 5 * 2 \hat{} n$
by (*simp add*: *power-add*)
finally have *mersenne-mod* (*mersenne-mod k n*) *n* $< 2 \hat{} n + 5$
by (*intro mersenne-mod-less′*) *auto*
thus *?thesis* **by** (*simp add*: *mersenne-mod2-def*)
qed

Since we subtract 2 at one point, the intermediate results can become negative. This is not a problem since our reduction modulo $2^p - 1$ happens to make them positive again immediately.

lemma *mersenne-mod-nonneg-strong*:
‹*mersenne-mod a p* ≥ 0› **if** ‹$- (2 \hat{} p) + 1 < a$›
proof (*cases* ‹$a < 0$›)
case *False*
with *that* **show** *?thesis*
by *simp*
next
case *True*
have ‹$- a$ *div* $- (2 \hat{} p) = - 1$›
by (*rule div-pos-neg-trivial*) (*use* ‹$a < 0$› *that* **in** *simp-all*)
then have ‹a *div* $2 \hat{} p = -1$›
by *simp*
moreover have ‹$- a$ *mod* $- (2 \hat{} p) = - a + - (2 \hat{} p)$›
by (*rule mod-pos-neg-trivial*) (*use* ‹$a < 0$› *that* **in** *simp-all*)
then have ‹a *mod* $2 \hat{} p = a + 2 \hat{} p$›
by *simp*
ultimately have ‹*mersenne-mod a p* $= a + 2 \hat{} p - 1$›
by (*simp add*: *mersenne-mod-def*)
also have ‹$\ldots > 0$› **using** *that* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *mersenne-mod2-nonneg-strong*:
assumes $a > -(2 \hat{} p) + 1$
shows *mersenne-mod2 a p* ≥ 0
unfolding *mersenne-mod2-def*
by (*rule mersenne-mod-nonneg*, *rule mersenne-mod-nonneg-strong*) (*use assms* **in** *auto*)

3.2 Efficient code for the Lucas–Lehmer sequence

```
primrec gen-lucas-lehmer-sequence'' :: nat ⇒ int ⇒ nat ⇒ int where
  gen-lucas-lehmer-sequence'' p a 0 = a
| gen-lucas-lehmer-sequence'' p a (Suc n) =
    gen-lucas-lehmer-sequence'' p (mersenne-mod2 (a ^ 2 − 2) p) n

lemma gen-lucas-lehmer-sequence''-correct:
  assumes [a = a'] (mod (2 ^ p − 1))
  shows   [gen-lucas-lehmer-sequence'' p a n = gen-lucas-lehmer-sequence a' n]
(mod (2 ^ p − 1))
  using assms
proof (induction n arbitrary: a a')
  case (Suc n)
  have [mersenne-mod2 (a ^ 2 − 2) p = a ^ 2 − 2] (mod (2 ^ p − 1))
    by (rule cong-mersenne-mod2)
  also have [a ^ 2 − 2 = a' ^ 2 − 2] (mod (2 ^ p − 1))
    by (intro cong-pow cong-diff Suc.prems cong-refl)
  finally have [gen-lucas-lehmer-sequence'' p (mersenne-mod2 (a^2 − 2) p) n =
                gen-lucas-lehmer-sequence (a'^2 − 2) n] (mod 2 ^ p − 1)
    by (rule Suc.IH)
  thus ?case
   by (auto simp del: gen-lucas-lehmer-sequence.simps simp: gen-lucas-lehmer-sequence-Suc')
qed auto

lemma gen-lucas-lehmer-sequence''-bounds:
  assumes a ≥ 0 a < 2 ^ p + 5 p > 2
  shows   gen-lucas-lehmer-sequence'' p a n ∈ {0..<2 ^ p + 5}
  using assms
proof (induction n arbitrary: a)
  case (Suc n)
  from Suc.prems have a ^ 2 < (2 ^ p + 5) ^ 2
    by (intro power-strict-mono Suc.prems) auto
  also have ... ≤ (2 ^ (p + 1)) ^ 2
    using power-increasing[of 3 p 2 :: int] ‹p > 2› by (intro power-mono) auto
  finally have a ^ 2 − 2 < 2 ^ (2 * p + 2)
    by (simp flip: power-mult mult-ac)
  moreover {
    from ‹p > 2› have (2 ^ p) ≥ (2 ^ 3 :: int)
      by (intro power-increasing) auto
    hence −(2 ^ p) + 1 < (−2 :: int)
      by simp
    also have −2 ≤ a ^ 2 − 2
      by simp
    finally have mersenne-mod2 (a ^ 2 − 2) p ≥ 0
      by (rule mersenne-mod2-nonneg-strong)
  }
  ultimately have gen-lucas-lehmer-sequence'' p (mersenne-mod2 (a^2 − 2) p) n
∈ {0..<2 ^ p + 5}
    using ‹p > 2› by (intro Suc.IH mersenne-mod2-less) auto
```

thus *?case* **by** *simp*
qed *auto*

3.3 Code for the Lucas–Lehmer test

lemmas [*code del*] = *lucas-lehmer-test-code-arithmetic*

lemma *lucas-lehmer-test-code* [*code*]:
lucas-lehmer-test p =
(*2 < p* ∧ (*let x* = *gen-lucas-lehmer-sequence'' p 4* (*p* − *2*) *in x* = *0* ∨ *x* = (*push-bit p 1*) − *1*))
unfolding *lucas-lehmer-test-def*
proof (*rule conj-cong*)
assume *p* > *2*
define *x* **where** *x* = *gen-lucas-lehmer-sequence'' p 4* (*p* − *2*)
from ‹*p* > *2*› **have** *2* ^ *3* ≤ (*2* ^ *p* :: *int*) **by** (*intro power-increasing*) *auto*
hence *2* ^ *p* ≥ (*8* :: *int*) **by** *simp*
hence *bounds*: *x* ∈ {*0*..<*2* ^ *p* + *5*}
unfolding *x-def* **using** ‹*p* > *2*› **by** (*intro gen-lucas-lehmer-sequence''-bounds*)
auto
have *2* ^ *p* − *1 dvd gen-lucas-lehmer-sequence 4* (*p* − *2*) ⟷ *2* ^ *p* − *1 dvd x*
unfolding *x-def* **by** (*intro cong-dvd-iff cong-sym*[*OF gen-lucas-lehmer-sequence''-correct*])
auto
also have ... ⟷ *x* ∈ {*0*, *2* ^ *p* − *1*}
proof
assume *2* ^ *p* − *1 dvd x*
then obtain *k* **where** *k*: *x* = (*2* ^ *p* − *1*) ∗ *k* **by** *auto*
have *k* ≥ *0* **using** *bounds* ‹*2* ^ *p* ≥ *8*›
by (*auto simp*: *k zero-le-mult-iff*)
moreover {
have *x* < *2* ^ *p* + *5* **using** *bounds* **by** *simp*
also have ... ≤ (*2* ^ *p* − *1*) ∗ *2*
using ‹*2* ^ *p* ≥ *8*› **by** *simp*
finally have (*2* ^ *p* − *1*) ∗ *k* < (*2* ^ *p* − *1*) ∗ *2*
unfolding *k* **.**
hence *k* < *2*
by (*subst* (*asm*) *mult-less-cancel-left*) *auto*
}
ultimately have *k* = *0* ∨ *k* = *1* **by** *auto*
thus *x* ∈ {*0*, *2* ^ *p* − *1*}
using *k* **by** *auto*
qed *auto*
finally show (*2* ^ *p* − *1 dvd gen-lucas-lehmer-sequence 4* (*p* − *2*)) =
((*let x* = *x in x* = *0* ∨ *x* = (*push-bit p 1*) − *1*))
by (*simp add*: *Let-def push-bit-eq-mult*)
qed *auto*

3.4 Examples

Note that for some reason, the clever bit-arithmetic version of the Lucas–Lehmer test is actually much slower than the one using integer arithmetic when using PolyML, and even more so when using the built-in evaluator in Isabelle (which also uses PolyML with a slightly different setup).

I do not quite know why this is the case, but it is likely because of inefficient implementations of bit arithmetic operations in PolyML and/or the code generator setup for it.

When running with GHC, the bit-arithmetic version is *much* faster.

value *filter mersenne-prime* [*0..<100*]

lemma *prime* (*2 ^ 521 − 1* :: *nat*)
  by (*subst lucas-lehmer-correct'*) *eval*

lemma *prime* (*2 ^ 4253 − 1* :: *nat*)
  by (*subst lucas-lehmer-correct'*) *eval*

end

References

[1] J. W. Bruce. A really trivial proof of the Lucas-Lehmer test. *The American Mathematical Monthly*, 100(4):370–371, 1993.

[2] Ö. J. Rödseth. A note on primality tests for $n = h \cdot 2^n - 1$. *BIT Numerical Mathematics*, 34(3):451–454, Sep 1994.

[3] Wikipedia contributors. Lucas–Lehmer primality test — Wikipedia, the free encyclopedia, 2020. [Online; accessed 17 Jan 2020].

[4] Wikipedia contributors. Mersenne prime — Wikipedia, the free encyclopedia, 2020. [Online; accessed 17 Jan 2020].